CHAPTER 7

LOGICAL AGENTS

In which we design agents that can form representations of a complex world, use a process of inference to derive new representations about the world, and use these new representations to deduce what to do.

Knowledge-based agents
Reasoning
Representation

Humans, it seems, know things; and what they know helps them do things. In AI, **knowledge-based agents** use a process of **reasoning** over an internal **representation** of knowledge to decide what actions to take.

The problem-solving agents of Chapters 3 and 4 know things, but only in a very limited, inflexible sense. They know what actions are available and what the result of performing a specific action from a specific state will be, but they don't know general facts. A route-finding agent doesn't know that it is impossible for a road to be a negative number of kilometers long. An 8-puzzle agent doesn't know that two tiles cannot occupy the same space. The knowledge they have is very useful for finding a path from the start to a goal, but not for anything else.

The atomic representations used by problem-solving agents are also very limiting. In a partially observable environment, for example, a problem-solving agent's only choice for representing what it knows about the current state is to list all possible concrete states. I could give a human the goal of driving to a U.S. town with population less than 10,000, but to say that to a problem-solving agent, I could formally describe the goal only as an explicit set of the 16,000 or so towns that satisfy the description.

Chapter 6 introduced our first factored representation, whereby states are represented as assignments of values to variables; this is a step in the right direction, enabling some parts of the agent to work in a domain-independent way and allowing for more efficient algorithms. In this chapter, we take this step to its logical conclusion, so to speak—we develop **logic** as a general class of representations to support knowledge-based agents. These agents can combine and recombine information to suit myriad purposes. This can be far removed from the needs of the moment—as when a mathematician proves a theorem or an astronomer calculates the Earth's life expectancy. Knowledge-based agents can accept new tasks in the form of explicitly described goals; they can achieve competence quickly by being told or learning new knowledge about the environment; and they can adapt to changes in the environment by updating the relevant knowledge.

We begin in Section 7.1 with the overall agent design. Section 7.2 introduces a simple new environment, the wumpus world, and illustrates the operation of a knowledge-based agent without going into any technical detail. Then we explain the general principles of **logic** in Section 7.3 and the specifics of **propositional logic** in Section 7.4. Propositional logic is a factored representation; while less expressive than **first-order logic** (Chapter 8), which is the canonical structured representation, propositional logic illustrates all the basic concepts

---

function KB-AGENT(*percept*) **returns** an *action*
  persistent: *KB*, a knowledge base
              t, a counter, initially 0, indicating time

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  action ← ASK(*KB*, MAKE-ACTION-QUERY(*t*))
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  t ← *t* + 1
  return *action*

Figure 7.1 A generic knowledge-based agent. Given a percept, the agent adds the percept to its knowledge base, asks the knowledge base for the best action, and tells the knowledge base that it has in fact taken that action.

---

of logic. It also comes with well-developed inference technologies, which we describe in sections 7.5 and 7.6. Finally, Section 7.7 combines the concept of knowledge-based agents with the technology of propositional logic to build some simple agents for the wumpus world.

7.1 Knowledge-Based Agents

The central component of a knowledge-based agent is its **knowledge base**, or KB. A knowledge base is a set of **sentences**. (Here "sentence" is used as a technical term. It is related but not identical to the sentences of English and other natural languages.) Each sentence is expressed in a language called a **knowledge representation language** and represents some assertion about the world. When the sentence is taken as being given without being derived from other sentences, we call it an **axiom**.

Knowledge base

Sentence

Knowledge representation language

Axiom

There must be a way to add new sentences to the knowledge base and a way to query what is known. The standard names for these operations are TELL and ASK, respectively. Both operations may involve **inference**—that is, deriving new sentences from old. Inference must obey the requirement that when one ASKs a question of the knowledge base, the answer should follow from what has been told (or TELLed) to the knowledge base previously. Later in this chapter, we will be more precise about the crucial word "follow." For now, take it to mean that the inference process should not make things up as it goes along.

Inference

Figure 7.1 shows the outline of a knowledge-based agent program. Like all our agents, it takes a percept as input and returns an action. The agent maintains a knowledge base, *KB*, which may initially contain some **background knowledge**.

Background knowledge

Each time the agent program is called, it does three things. First, it TELLs the knowledge base what it perceives. Second, it ASKs the knowledge base what action it should perform. In the process of answering this query, extensive reasoning may be done about the current state of the world, about the outcomes of possible action sequences, and so on. Third, the agent program TELLs the knowledge base which action was chosen, and returns the action so that it can be executed.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators on one side and the core representation and reasoning system on the other. MAKE-PERCEPT-SENTENCE constructs a sentence as-

serting that the agent perceived the given percept at the given time. MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time. Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed. The details of the inference mechanisms are hidden inside TELL and ASK. Later sections will reveal these details.

The agent in Figure 7.1 appears quite similar to the agents with internal state described in Chapter 2. Because of the definitions of TELL and ASK, however, the knowledge-based agent is not an arbitrary program for calculating actions. It is amenable to a description at the **knowledge level**, where we need specify only what the agent knows and what its goals are, in order to determine its behavior.

For example, an automated taxi might have the goal of taking a passenger from San Francisco to Marin County and might know that the Golden Gate Bridge is the only link between the two locations. Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal*. Notice that this analysis is independent of how the taxi works at the **implementation level**. It doesn't matter whether its geographical knowledge is implemented as linked lists or pixel maps, or whether it reasons by manipulating strings of symbols stored in registers or by propagating noisy signals in a network of neurons.

A knowledge-based agent can be built simply by TELLing it what it needs to know. Starting with an empty knowledge base, the agent designer can TELL sentences one by one until the agent knows how to operate in its environment. This is called the **declarative** approach to system building. In contrast, the **procedural** approach encodes desired behaviors directly as program code. In the 1970s and 1980s, advocates of the two approaches engaged in heated debates. We now understand that a successful agent often combines both declarative and procedural elements in its design, and that declarative knowledge can often be compiled into more efficient procedural code.

We can also provide a knowledge-based agent with mechanisms that allow it to learn for itself. These mechanisms, which are discussed in Chapter 19, create general knowledge about the environment from a series of percepts. A learning agent can be fully autonomous.

Knowledge level

Implementation level

Declarative
Procedural

7.2 The Wumpus World

Wumpus world

In this section we describe an environment in which knowledge-based agents can show their worth. The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the terrible wumpus, a beast that eats anyone who enters its room. The wumpus can be shot by an agent, but the agent has only one arrow. Some rooms contain bottomless pits that will trap anyone who wanders into these rooms (except for the wumpus, which is too big to fall in). The only redeeming feature of this bleak environment is the possibility of finding a heap of gold. Although the wumpus world is rather tame by modern computer game standards, it illustrates some important points about intelligence.

A sample wumpus world is shown in Figure 7.2. The precise definition of the task environment is given, as suggested in Section 2.3, by the PEAS description:

- **Performance measure**: +1000 for climbing out of the cave with the gold, −1000 for falling into a pit or being eaten by the wumpus, −1 for each action taken, and −10 for using up the arrow. The game ends either when the agent dies or when the agent climbs out of the cave.
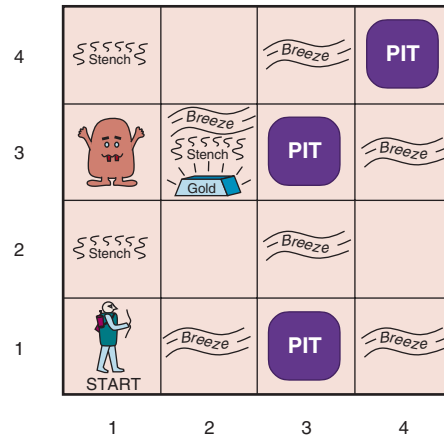
Figure 7.2 A typical wumpus world. The agent is in the bottom left corner, facing east (rightward).

- **Environment**: A 4×4 grid of rooms, with walls surrounding the grid. The agent always starts in the square labeled [1,1], facing to the east. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square. In addition, each square other than the start can be a pit, with probability 0.2.
- **Actuators**: The agent can move *Forward*, *TurnLeft* by 90°, or *TurnRight* by 90°. The agent dies a miserable death if it enters a square containing a pit or a live wumpus. (It is safe, albeit smelly, to enter a square with a dead wumpus.) If an agent tries to move forward and bumps into a wall, then the agent does not move. The action *Grab* can be used to pick up the gold if it is in the same square as the agent. The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (and hence kills) the wumpus or hits a wall. The agent has only one arrow, so only the first *Shoot* action has any effect. Finally, the action *Climb* can be used to climb out of the cave, but only from square [1,1].
- **Sensors**: The agent has five sensors, each of which gives a single bit of information:
    - In the squares directly (not diagonally) adjacent to the wumpus, the agent will perceive a *Stench*.[1]
    - In the squares directly adjacent to a pit, the agent will perceive a *Breeze*.
    - In the square where the gold is, the agent will perceive a *Glitter*.
    - When an agent walks into a wall, it will perceive a *Bump*.
    - When the wumpus is killed, it emits a woeful *Scream* that can be perceived anywhere in the cave.

    The percepts will be given to the agent program in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent program will get [*Stench*, *Breeze*, *None*, *None*, *None*].

---

[1] Presumably the square containing the wumpus also has a stench, but any agent entering that square is eaten before being able to perceive anything.
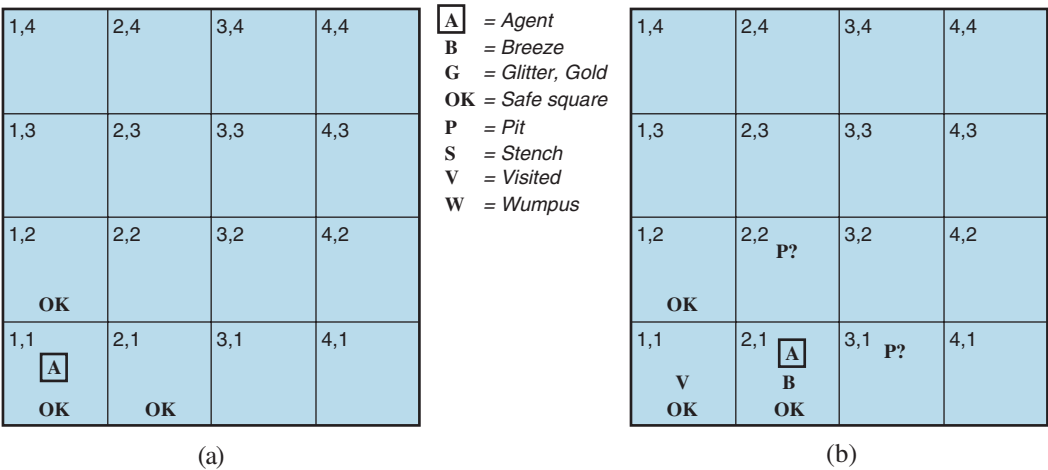
1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2	3,2	4,2
1,1 A OK	2,1 OK	3,1	4,1

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2 OK	2,2 P?	3,2	4,2
1,1 V OK	2,1 A B OK	3,1 P?	4,1

(a)   (b)

Figure 7.3 The first step taken by the agent in the wumpus world. (a) The initial situation, after percept [*None,None,None,None,None*]. (b) After moving to [2,1] and perceiving [*None,Breeze,None,None,None*].

We can characterize the wumpus environment along the various dimensions given in Chapter 2. Clearly, it is deterministic, discrete, static, and single-agent. (The wumpus doesn't move, fortunately.) It is sequential, because rewards may come only after many actions are taken. It is partially observable, because some aspects of the state are not directly perceivable: the agent's location, the wumpus's state of health, and the availability of an arrow. As for the locations of the pits and the wumpus: we could treat them as unobserved parts of the state—in which case, the transition model for the environment is completely known, and finding the locations of pits completes the agent's knowledge of the state. Alternatively, we could say that the transition model itself is unknown because the agent doesn't know which *Forward* actions are fatal—in which case, discovering the locations of pits and wumpus completes the agent's knowledge of the transition model.

For an agent in the environment, the main challenge is its initial ignorance of the configuration of the environment; overcoming this ignorance seems to require logical reasoning. In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely. Occasionally, the agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure 7.2. We use an informal knowledge representation language consisting of writing down symbols in a grid (as in Figures 7.3 and 7.4).

The agent's initial knowledge base contains the rules of the environment, as described previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square; we denote that with an "A" and "OK," respectively, in square [1,1].

The first percept is [*None,None,None,None,None*], from which the agent can conclude that its neighboring squares, [1,2] and [2,1], are free of dangers—they are OK. Figure 7.3(a) shows the agent's state of knowledge at this point.

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 [A] S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

[A] = Agent
B   = Breeze
G   = Glitter, Gold
OK = Safe square
P   = Pit
S   = Stench
V   = Visited
W = Wumpus

1,4	2,4 P?	3,4	4,4
1,3 W!	2,3 [A] S G B	3,3 P?	4,3
1,2 S V OK	2,2 V OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

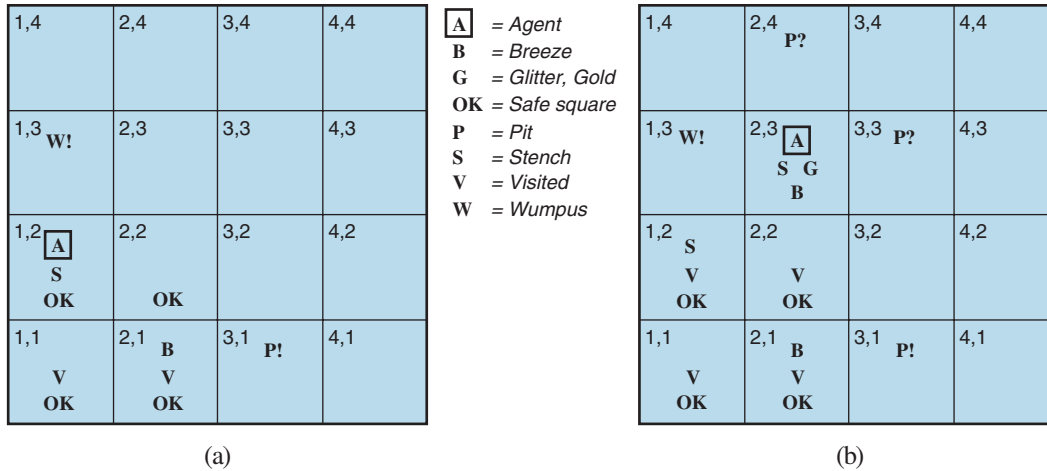(a)   (b)

Figure 7.4 Two later stages in the progress of the agent. (a) After moving to [1,1] and then [1,2], and perceiving [*Stench,None,None,None,None*]. (b) After moving to [2,2] and then [2,3], and perceiving [*Stench,Breeze,Glitter,None,None*].

A cautious agent will move only into a square that it knows to be OK. Let us suppose the agent decides to move forward to [2,1]. The agent perceives a breeze (denoted by "B") in [2,1], so there must be a pit in a neighboring square. The pit cannot be in [1,1], by the rules of the game, so there must be a pit in [2,2] or [3,1] or both. The notation "P?" in Figure 7.3(b) indicates a possible pit in those squares. At this point, there is only one known square that is OK and that has not yet been visited. So the prudent agent will turn around, go back to [1,1], and then proceed to [1,2].

The agent perceives a stench in [1,2], resulting in the state of knowledge shown in Figure 7.4(a). The stench in [1,2] means that there must be a wumpus nearby. But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2] (or the agent would have detected a stench when it was in [2,1]). Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this inference. Moreover, the lack of a breeze in [1,2] implies that there is no pit in [2,2]. Yet the agent has already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1]. This is a fairly difficult inference, because it combines knowledge gained at different times in different places and relies on the lack of a percept to make one crucial step.

The agent has now proved to itself that there is neither a pit nor a wumpus in [2,2], so it is OK to move there. We do not show the agent's state of knowledge at [2,2]; we just assume that the agent turns and moves to [2,3], giving us Figure 7.4(b). In [2,3], the agent detects a glitter, so it should grab the gold and then return home.

Note that in each case for which the agent draws a conclusion from the available information, that conclusion is *guaranteed* to be correct if the available information is correct. This is a fundamental property of logical reasoning. In the rest of this chapter, we describe how to build logical agents that can represent information and draw conclusions such as those described in the preceding paragraphs.

7.3 Logic

This section summarizes the fundamental concepts of logical representation and reasoning. These beautiful ideas are independent of any of logic's particular forms. We therefore postpone the technical details of those forms until the next section, using instead the familiar example of ordinary arithmetic.

In Section 7.1, we said that knowledge bases consist of sentences. These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed. The notion of syntax is clear enough in ordinary arithmetic: "$x + y = 4$" is a well-formed sentence, whereas "$x4y+ =$" is not.

A logic must also define the **semantics**, or meaning, of sentences. The semantics defines the **truth** of each sentence with respect to each **possible world**. For example, the semantics for arithmetic specifies that the sentence "$x + y = 4$" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1. In standard logics, every sentence must be either true or false in each possible world—there is no "in between."[2]

When we need to be precise, we use the term **model** in place of "possible world." Whereas possible worlds might be thought of as (potentially) real environments that the agent might or might not be in, models are mathematical abstractions, each of which has a fixed truth value (true or false) for every relevant sentence. Informally, we may think of a possible world as, for example, having x men and y women sitting at a table playing bridge, and the sentence $x + y = 4$ is true when there are four people in total. Formally, the possible models are just all possible assignments of nonnegative integers to the variables x and y. Each such assignment determines the truth of any sentence of arithmetic whose variables are x and y. If a sentence α is true in model m, we say that m **satisfies** α or sometimes m **is a model of** α. We use the notation $M(\alpha)$ to mean the set of all models of α.

Now that we have a notion of truth, we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences—the idea that a sentence *follows logically* from another sentence. In mathematical notation, we write

$$\alpha \models \beta$$

to mean that the sentence α entails the sentence β. The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which α is true, β is also true. Using the notation just introduced, we can write

$$\alpha \models \beta \text{ if and only if } M(\alpha) \subseteq M(\beta).$$

(Note the direction of the $\subseteq$ here: if $\alpha \models \beta$, then α is a *stronger* assertion than β: it rules out *more* possible worlds.) The relation of entailment is familiar from arithmetic; we are happy with the idea that the sentence $x = 0$ entails the sentence $xy = 0$. Obviously, in any model where x is zero, it is the case that xy is zero (regardless of the value of y).

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 7.3(b): the agent has detected nothing in [1,1] and a breeze in [2,1]. These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB. The agent is interested in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits. Each of the three squares might or might

Margin terms: Syntax · Semantics · Truth · Possible world · Model · Satisfaction · Entailment

---

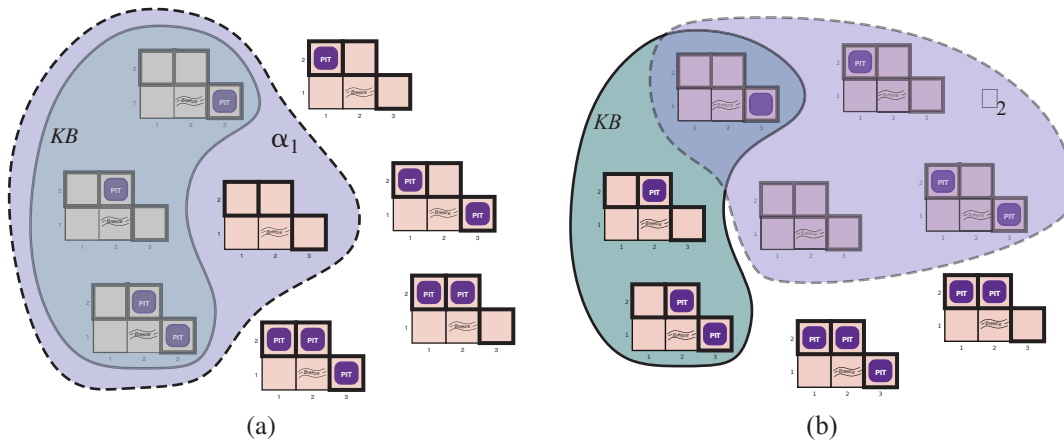[2] **Fuzzy logic**, discussed in Chapter 13, allows for degrees of truth.

Figure 7.5 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α_1 (no pit in [1,2]). (b) Dotted line shows models of α_2 (no pit in [2,2]).

not contain a pit, so (ignoring other aspects of the world for now) there are $2^3 = 8$ possible models. These eight models are shown in Figure 7.5.[3]

The KB can be thought of as a set of sentences or as a single sentence that asserts all the individual sentences. The KB is false in models that contradict what the agent knows— for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown surrounded by a solid line in Figure 7.5. Now let us consider two possible conclusions:

α_1 = "There is no pit in [1,2]." α_2 = "There is no pit in [2,2]."

We have surrounded the models of α_1 and α_2 with dotted lines in Figures 7.5(a) and 7.5(b), respectively. By inspection, we see the following:

in every model in which *KB* is true, α_1 is also true.

Hence, $KB \models \alpha_1$: there is no pit in [1,2]. We can also see that

in some models in which *KB* is true, α_2 is false.

Hence, *KB* does not entail α_2: the agent *cannot* conclude that there is no pit in [2,2]. (Nor can it conclude that there *is* a pit in [2,2].)[4]

The preceding example not only illustrates entailment but also shows how the definition of entailment can be applied to derive conclusions—that is, to carry out **logical inference**. The inference algorithm illustrated in Figure 7.5 is called **model checking**, because it enumerates all possible models to check that α is true in all models in which *KB* is true, that is, that $M(KB) \subseteq M(\alpha)$.

Logical inference

Model checking

---

[3] Although the figure shows the models as partial wumpus worlds, they are really nothing more than assignments of *true* and *false* to the sentences "there is a pit in [1,2]" etc. Models, in the mathematical sense, do not need to have 'orrible 'airy wumpuses in them.

[4] The agent can calculate the *probability* that there is a pit in [2,2]; Chapter 12 shows how.
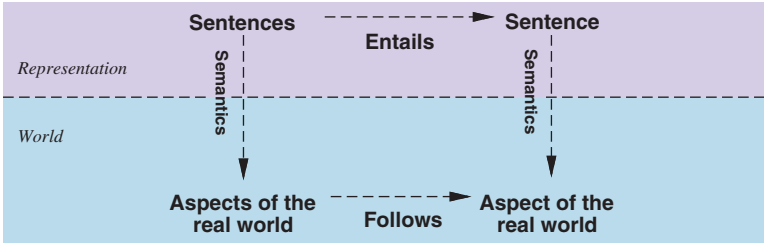
Figure 7.6 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

In understanding entailment and inference, it might help to think of the set of all consequences of *KB* as a haystack and of α as a needle. Entailment is like the needle being in the haystack; inference is like finding it. This distinction is embodied in some formal notation: if an inference algorithm *i* can derive α from *KB*, we write

$$KB \vdash_i \alpha,$$

which is pronounced "α is derived from *KB* by *i*" or "*i* derives α from *KB*."

Sound
Truth-preserving
An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving**. Soundness is a highly desirable property. An unsound inference procedure essentially makes things up as it goes along—it announces the discovery of nonexistent needles. It is easy to see that model checking, when it is applicable,[5] is a sound procedure.

Completeness
The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed. For real haystacks, which are finite in extent, it seems obvious that a systematic examination can always decide whether the needle is in the haystack. For many knowledge bases, however, the haystack of consequences is infinite, and completeness becomes an important issue.[6] Fortunately, there are complete inference procedures for logics that are sufficiently expressive to handle many knowledge bases.

We have described a reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, *if KB is true in the real world, then any sentence α derived from KB by a sound inference procedure is also true in the real world.* So, while an inference process operates on "syntax"—internal physical configurations such as bits in registers or patterns of electrical blips in brains—the process *corresponds* to the real-world relationship whereby some aspect of the real world is the case by virtue of other aspects of the real world being the case.[7] This correspondence between world and representation is illustrated in Figure 7.6.

Grounding
The final issue to consider is **grounding**—the connection between logical reasoning processes and the real environment in which the agent exists. In particular, *how do we know that*

---

[5] Model checking works if the space of models is finite—for example, in wumpus worlds of fixed size. For arithmetic, on the other hand, the space of models is infinite: even if we restrict ourselves to the integers, there are infinitely many pairs of values for *x* and *y* in the sentence $x + y = 4$.

[6] Compare with the case of infinite search spaces in Chapter 3, where depth-first search is not complete.

[7] As Wittgenstein (1922) put it in his famous *Tractatus*: "The world is everything that is the case."

KB is true in the real world? (After all, *KB* is just "syntax" inside the agent's head.) This is a philosophical question about which many, many books have been written. (See Chapter 27.) A simple answer is that the agent's sensors create the connection. For example, our wumpus-world agent has a smell sensor. The agent program creates a suitable sentence whenever there is a smell. Then, whenever that sentence is in the knowledge base, it is true in the real world. Thus, the meaning and truth of percept sentences are defined by the processes of sensing and sentence construction that produce them. What about the rest of the agent's knowledge, such as its belief that wumpuses cause smells in adjacent squares? This is not a direct representation of a single percept, but a general rule—derived, perhaps, from perceptual experience but not identical to a statement of that experience. General rules like this are produced by a sentence construction process called **learning**, which is the subject of Part V. Learning is fallible. It could be the case that wumpuses cause smells *except on February 29 in leap years*, which is when they take their baths. Thus, *KB* may not be true in the real world, but with good learning procedures, there is reason for optimism.

7.4 Propositional Logic: A Very Simple Logic

We now present **propositional logic**. We describe its syntax (the structure of sentences) and its semantics (the way in which the truth of sentences is determined). From these, we derive a simple, syntactic algorithm for logical inference that implements the semantic notion of entailment. Everything takes place, of course, in the wumpus world.

Propositional logic

7.4.1 Syntax

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** consist of a single **proposition symbol**. Each such symbol stands for a proposition that can be true or false. We use symbols that start with an uppercase letter and may contain other letters or subscripts, for example: P, Q, R, $W_{1,3}$ and *FacingEast*. The names are arbitrary but are often chosen to have some mnemonic value—we use $W_{1,3}$ to stand for the proposition that the wumpus is in [1,3]. (Remember that symbols such as $W_{1,3}$ are *atomic*, i.e., W, 1, and 3 are not meaningful parts of the symbol.) There are two proposition symbols with fixed meanings: *True* is the always-true proposition and *False* is the always-false proposition. **Complex sentences** are constructed from simpler sentences, using parentheses and operators called **logical connectives**. There are five connectives in common use:

Atomic sentences
Proposition symbol

Complex sentences
Logical connectives

> ¬ (not). A sentence such as $\neg W_{1,3}$ is called the **negation** of $W_{1,3}$. A **literal** is either an atomic sentence (a **positive literal**) or a negated atomic sentence (a **negative literal**).

Negation
Literal

> ∧ (and). A sentence whose main connective is ∧, such as $W_{1,3} \wedge P_{3,1}$, is called a **conjunction**; its parts are the **conjuncts**. (The ∧ looks like an "A" for "And.")

Conjunction

> ∨ (or). A sentence whose main connective is ∨, such as $(W_{1,3} \wedge P_{3,1}) \vee W_{2,2}$, is a **disjunction**; its parts are **disjuncts**—in this example, $(W_{1,3} \wedge P_{3,1})$ and $W_{2,2}$.

Disjunction

> ⇒ (implies). A sentence such as $(W_{1,3} \wedge P_{3,1}) \Rightarrow \neg W_{2,2}$ is called an **implication** (or conditional). Its **premise** or **antecedent** is $(W_{1,3} \wedge P_{3,1})$, and its **conclusion** or **consequent** is $\neg W_{2,2}$. Implications are also known as **rules** or **if–then** statements. The implication symbol is sometimes written in other books as ⊃ or →.

Implication
Premise
Conclusion
Rules

> ⇔ (if and only if). The sentence $W_{1,3} \Leftrightarrow \neg W_{2,2}$ is a **biconditional**.

Biconditional

$$
\begin{aligned}
\textit{Sentence} \;&\rightarrow\; \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} \;&\rightarrow\; \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \ldots \\
\textit{ComplexSentence} \;&\rightarrow\; (\,\textit{Sentence}\,) \\
&\mid\quad \neg\, \textit{Sentence} \\
&\mid\quad \textit{Sentence} \wedge \textit{Sentence} \\
&\mid\quad \textit{Sentence} \vee \textit{Sentence} \\
&\mid\quad \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid\quad \textit{Sentence} \Leftrightarrow \textit{Sentence}
\end{aligned}
$$

OPERATOR PRECEDENCE   : $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Figure 7.7 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Figure 7.7 gives a formal grammar of propositional logic. (BNF notation is explained on page 1030.) The BNF grammar is augmented with an operator precedence list to remove ambiguity when multiple operators are used. The "not" operator ($\neg$) has the highest precedence, which means that in the sentence $\neg A \wedge B$ the $\neg$ binds most tightly, giving us the equivalent of $(\neg A) \wedge B$ rather than $\neg(A \wedge B)$. (The notation for ordinary arithmetic is the same: $-2 + 4$ is 2, not –6.) When appropriate, we also use parentheses and square brackets to clarify the intended sentence structure and improve readability.

7.4.2 Semantics

Having specified the syntax of propositional logic, we now specify its semantics. The semantics defines the rules for determining the truth of a sentence with respect to a particular model. In propositional logic, a model simply sets the **truth value**—*true* or *false*—for every proposition symbol. For example, if the sentences in the knowledge base make use of the proposition symbols $P_{1,2}$, $P_{2,2}$, and $P_{3,1}$, then one possible model is

$$m_1 = \{P_{1,2}=false,\ P_{2,2}=false,\ P_{3,1}=true\}.$$

With three proposition symbols, there are $2^3 = 8$ possible models—exactly those depicted in Figure 7.5. Notice, however, that the models are purely mathematical objects with no necessary connection to wumpus worlds. $P_{1,2}$ is just a symbol; it might mean "there is a pit in [1,2]" or "I'm in Paris today and tomorrow."

The semantics for propositional logic must specify how to compute the truth value of *any* sentence, given a model. This is done recursively. All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives. Atomic sentences are easy:

- *True* is true in every model and *False* is false in every model.
- The truth value of every other proposition symbol must be specified directly in the model. For example, in the model m_1 given earlier, $P_{1,2}$ is false.

Truth value

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is *true* and Q is *false* (the third row). Then look in that row under the $P \vee Q$ column to see the result: *true*.

For complex sentences, we have five rules, which hold for any subsentences P and Q (atomic or complex) in any model m (here "iff" means "if and only if"):

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.

The rules can also be expressed with **truth tables** that specify the truth value of a complex   Truth table
sentence for each possible assignment of truth values to its components. Truth tables for the five connectives are given in Figure 7.8. From these tables, the truth value of any sentence s can be computed with respect to any model m by a simple recursive evaluation. For example, the sentence $\neg P_{1,2} \wedge (P_{2,2} \vee P_{3,1})$, evaluated in m_1, gives *true* $\wedge$ (*false* $\vee$ *true*) $=$ *true* $\wedge$ *true* $=$ *true*. Exercise 7.TRUV asks you to write the algorithm PL-TRUE?(s, m), which computes the truth value of a propositional logic sentence s in a model m.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that $P \vee Q$ is true when P is true or Q is true *or both*. A different connective, called "exclusive or" ("xor" for short), yields false when both disjuncts are true.[8] There is no consensus on the symbol for exclusive or; some choices are $\dot\vee$ or $\neq$ or $\oplus$.

The truth table for $\Rightarrow$ may not quite fit one's intuitive understanding of "P implies Q" or "if P then Q." For one thing, propositional logic does not require any relation of *causation* or *relevance* between P and Q. The sentence "5 is odd implies Tokyo is the capital of Japan" is a true sentence of propositional logic (under the normal interpretation), even though it is a decidedly odd sentence of English. Another point of confusion is that any implication is true whenever its antecedent is false. For example, "5 is even implies Sam is smart" is true, regardless of whether Sam is smart. This seems bizarre, but it makes sense if you think of "$P \Rightarrow Q$" as saying, "If P is true, then I am claiming that Q is true; otherwise I am making no claim." The only way for this sentence to be *false* is if P is true but Q is false.

The biconditional, $P \Leftrightarrow Q$, is true whenever both $P \Rightarrow Q$ and $Q \Rightarrow P$ are true. In English, this is often written as "P if and only if Q." Many of the rules of the wumpus world are best

---

[8] Latin uses two separate words: "vel" is inclusive or and "aut" is exclusive or.

written using $\Leftrightarrow$. For example, a square is breezy *if* a neighboring square has a pit, and a square is breezy *only if* a neighboring square has a pit. So we need a biconditional,

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1}),$$

where $B_{1,1}$ means that there is a breeze in [1,1].

7.4.3 A simple knowledge base

Now that we have defined the semantics for propositional logic, we can construct a knowledge base for the wumpus world. We focus first on the *immutable* aspects of the wumpus world, leaving the mutable aspects for a later section. For now, we need the following symbols for each $[x,y]$ location:

$P_{x,y}$ is true if there is a pit in $[x,y]$.
$W_{x,y}$ is true if there is a wumpus in $[x,y]$, dead or alive.
$B_{x,y}$ is true if there is a breeze in $[x,y]$.
$S_{x,y}$ is true if there is a stench in $[x,y]$.
$L_{x,y}$ is true if the agent is in location $[x,y]$.

The sentences we write will suffice to derive $\neg P_{1,2}$ (there is no pit in [1,2]), as was done informally in Section 7.3. We label each sentence R_i so that we can refer to them:

- There is no pit in [1,1]:

    $R_1: \quad \neg P_{1,1}$.

- A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

    $R_2: \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1})$.
    $R_3: \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1})$.

- The preceding sentences are true in all wumpus worlds. Now we include the breeze percepts for the first two squares visited in the specific world the agent is in, leading up to the situation in Figure 7.3(b).

    $R_4: \quad \neg B_{1,1}$.
    $R_5: \quad B_{2,1}$.

7.4.4 A simple inference procedure

Our goal now is to decide whether $KB \models \alpha$ for some sentence α. For example, is $\neg P_{1,2}$ entailed by our KB? Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that α is true in every model in which KB is true. Models are assignments of *true* or *false* to every proposition symbol. Returning to our wumpus-world example, the relevant proposition symbols are $B_{1,1}$, $B_{2,1}$, $P_{1,1}$, $P_{1,2}$, $P_{2,1}$, $P_{2,2}$, and $P_{3,1}$. With seven symbols, there are $2^7 = 128$ possible models; in three of these, KB is true (Figure 7.9). In those three models, $\neg P_{1,2}$ is true, hence there is no pit in [1,2]. On the other hand, $P_{2,2}$ is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2].

Figure 7.9 reproduces in a more precise form the reasoning illustrated in Figure 7.5. A general algorithm for deciding entailment in propositional logic is shown in Figure 7.10. Like the BACKTRACKING-SEARCH algorithm on page 192, TT-ENTAILS? performs a recursive

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	false	false	false	false	false	false	true	true	true	true	false	false
false	false	false	false	false	false	true	true	true	false	true	false	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	true	false	false	false	false	false	true	true	false	true	true	false
false	true	false	false	false	false	true	true	true	true	true	true	_true_
false	true	false	false	false	true	false	true	true	true	true	true	_true_
false	true	false	false	false	true	true	true	true	true	true	true	_true_
false	true	false	false	true	false	false	true	false	false	true	true	false
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	true	true	true	true	true	true	false	true	true	false	true	false

Figure 7.9 A truth table constructed for the knowledge base given in the text. *KB* is true if R_1 through R_5 are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, $P_{1,2}$ is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

---

function TT-ENTAILS?(*KB*, α) **returns** *true* or *false*
  inputs: *KB*, the knowledge base, a sentence in propositional logic
        α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in *KB* and α
  return TT-CHECK-ALL(*KB*, α, *symbols*, { })

function TT-CHECK-ALL(*KB*, α, *symbols*, *model*) **returns** *true* or *false*
  if EMPTY?(*symbols*) **then**
      if PL-TRUE?(*KB*, *model*) **then return** PL-TRUE?(α, *model*)
      else return *true*   // when KB is false, always return true
  else
      P ← FIRST(*symbols*)
      rest ← REST(*symbols*)
      return (TT-CHECK-ALL(*KB*, α, *rest*, *model* ∪ {*P* = *true*})
            and
            TT-CHECK-ALL(*KB*, α, *rest*, *model* ∪ {*P* = *false* }))

Figure 7.10 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns *true* if a sentence holds within a model. The variable *model* represents a partial model—an assignment to some of the symbols. The keyword **and** here is an infix function symbol in the pseudocode programming language, not an operator in proposition logic; it takes two arguments and returns *true* or *false*.

enumeration of a finite space of assignments to symbols. The algorithm is **sound** because it implements directly the definition of entailment, and **complete** because it works for any *KB* and α and always terminates—there are only finitely many models to examine.

Of course, "finitely many" is not always the same as "few." If *KB* and α contain n symbols in all, then there are 2^n models. Thus, the time complexity of the algorithm is $O(2^n)$. (The space complexity is only $O(n)$ because the enumeration is depth-first.) Later in this chapter we show algorithms that are much more efficient in many cases. Unfortunately, propositional entailment is co-NP-complete (i.e., probably no easier than NP-complete—see Appendix A), so *every known inference algorithm for propositional logic has a worst-case complexity that is exponential in the size of the input.*

7.5 Propositional Theorem Proving

So far, we have shown how to determine entailment by *model checking*: enumerating models and showing that the sentence must hold in all models. In this section, we show how entailment can be done by **theorem proving**—applying rules of inference directly to the sentences in our knowledge base to construct a proof of the desired sentence without consulting models. If the number of models is large but the length of the proof is short, then theorem proving can be more efficient than model checking.

Before we plunge into the details of theorem-proving algorithms, we will need some additional concepts related to entailment. The first concept is **logical equivalence**: two sentences α and β are logically equivalent if they are true in the same set of models. We write this as $\alpha \equiv \beta$. (Note that $\equiv$ is used to make claims about sentences, while $\Leftrightarrow$ is used as part of a sentence.) For example, we can easily show (using truth tables) that $P \wedge Q$ and $Q \wedge P$ are logically equivalent; other equivalences are shown in Figure 7.11. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics. An alternative definition of equivalence is as follows: any two sentences α and β are equivalent if and only if each of them entails the other:

$$\alpha \equiv \beta \quad \text{if and only if} \quad \alpha \models \beta \text{ and } \beta \models \alpha.$$

The second concept we will need is **validity**. A sentence is valid if it is true in *all* models. For example, the sentence $P \vee \neg P$ is valid. Valid sentences are also known as **tautologies**—they are *necessarily* true. Because the sentence *True* is true in all models, every valid sentence is logically equivalent to *True*. What good are valid sentences? From our definition of entailment, we can derive the **deduction theorem**, which was known to the ancient Greeks:

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is valid.

(Exercise 7.DEDU asks for a proof.) Hence, we can decide if $\alpha \models \beta$ by checking that $(\alpha \Rightarrow \beta)$ is true in every model—which is essentially what the inference algorithm in Figure 7.10 does—or by proving that $(\alpha \Rightarrow \beta)$ is equivalent to *True*. Conversely, the deduction theorem states that every valid implication sentence describes a legitimate inference.

The final concept we will need is **satisfiability**. A sentence is satisfiable if it is true in, or satisfied by, *some* model. For example, the knowledge base given earlier, $(R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5)$, is satisfiable because there are three models in which it is true, as shown in Figure 7.9. Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence. The problem of determining the satisfiability of sentences

Theorem proving

Logical equivalence

Validity

Tautology

Deduction theorem

Satisfiability

$$(\alpha \wedge \beta) \equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge$$
$$(\alpha \vee \beta) \equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee$$
$$((\alpha \wedge \beta) \wedge \gamma) \equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge$$
$$((\alpha \vee \beta) \vee \gamma) \equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee$$
$$\neg(\neg\alpha) \equiv \alpha \quad \text{double-negation elimination}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition}$$
$$(\alpha \Rightarrow \beta) \equiv (\neg\alpha \vee \beta) \quad \text{implication elimination}$$
$$(\alpha \Leftrightarrow \beta) \equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination}$$
$$\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan}$$
$$\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan}$$
$$(\alpha \wedge (\beta \vee \gamma)) \equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee$$
$$(\alpha \vee (\beta \wedge \gamma)) \equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

in propositional logic—the **SAT** problem—was the first problem proved to be NP-complete. SAT
Many problems in computer science are really satisfiability problems. For example, all the constraint satisfaction problems in Chapter 6 ask whether the constraints are satisfiable by some assignment.

Validity and satisfiability are of course connected: α is valid iff $\neg\alpha$ is unsatisfiable; contrapositively, α is satisfiable iff $\neg\alpha$ is not valid. We also have the following useful result:

◀

$\alpha \models \beta$ *if and only if the sentence* $(\alpha \wedge \neg\beta)$ *is unsatisfiable.*

Proving β from α by checking the unsatisfiability of $(\alpha \wedge \neg\beta)$ corresponds exactly to the standard mathematical proof technique of *reductio ad absurdum* (literally, "reduction to an absurd thing"). It is also called proof by **refutation** or proof by **contradiction**. One assumes a sentence β to be false and shows that this leads to a contradiction with known axioms α. This contradiction is exactly what is meant by saying that the sentence $(\alpha \wedge \neg\beta)$ is unsatisfiable.

Reductio ad absurdum

Refutation

Contradiction

7.5.1 Inference and proofs

This section covers **inference rules** that can be applied to derive a **proof**—a chain of conclu-   Inference rules
sions that leads to the desired goal. The best-known rule is called **Modus Ponens** (Latin for   Proof
mode that affirms) and is written   Modus Ponens

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

The notation means that, whenever any sentences of the form $\alpha \Rightarrow \beta$ and α are given, then the sentence β can be inferred. For example, if $(WumpusAhead \wedge WumpusAlive) \Rightarrow Shoot$ and $(WumpusAhead \wedge WumpusAlive)$ are given, then *Shoot* can be inferred.

Another useful inference rule is **And-Elimination**, which says that, from a conjunction,   And-Elimination
any of the conjuncts can be inferred:

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from $(WumpusAhead \wedge WumpusAlive)$, *WumpusAlive* can be inferred.

By considering the possible truth values of α and β, one can easily show once and for all that Modus Ponens and And-Elimination are sound. These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

All of the logical equivalences in Figure 7.11 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)} \quad \text{and} \quad \frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain $\alpha \Rightarrow \beta$ and α from β.

Let us see how these inference rules and equivalences can be used in the wumpus world. We start with the knowledge base containing R_1 through R_5 and show how to prove $\neg P_{1,2}$, that is, there is no pit in [1,2]:

1. Apply biconditional elimination to R_2 to obtain

    R_6: $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.

2. Apply And-Elimination to R_6 to obtain

    R_7: $((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$.

3. Logical equivalence for contrapositives gives

    R_8: $(\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$.

4. Apply Modus Ponens with R_8 and the percept R_4 (i.e., $\neg B_{1,1}$), to obtain

    R_9: $\neg(P_{1,2} \vee P_{2,1})$.

5. Apply De Morgan's rule, giving the conclusion

    R_{10}: $\neg P_{1,2} \wedge \neg P_{2,1}$.

That is, neither [1,2] nor [2,1] contains a pit.

Any of the search algorithms in Chapter 3 can be used to find a sequence of steps that constitutes a proof like this. We just need to define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

Thus, searching for proofs is an alternative to enumerating models. In many practical cases *finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.* For example, the proof just given leading to $\neg P_{1,2} \wedge \neg P_{2,1}$ does not mention the propositions $B_{2,1}$, $P_{1,1}$, $P_{2,2}$, or $P_{3,1}$. They can be ignored because the goal proposition, $P_{1,2}$, appears only in sentence R_2; the other propositions in R_2 appear only in R_4 and R_2; so R_1, R_3, and R_5 have no bearing on the proof. The same would hold even if we added a million more sentences to the knowledge base; the simple truth-table algorithm, on the other hand, would be overwhelmed by the exponential explosion of models.

One final property of logical systems is **monotonicity**, which says that the set of en-
tailed sentences can only *increase* as information is added to the knowledge base.[9] For any
sentences α and β,

$$\text{if} \quad KB \models \alpha \quad \text{then} \quad KB \wedge \beta \models \alpha.$$

For example, suppose the knowledge base contains the additional assertion β stating that there
are exactly eight pits in the world. This knowledge might help the agent draw *additional* con-
clusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion
that there is no pit in [1,2]. Monotonicity means that inference rules can be applied whenever
suitable premises are found in the knowledge base—the conclusion of the rule must follow
regardless of what else is in the knowledge base.

7.5.2 Proof by resolution

We have argued that the inference rules covered so far are *sound*, but we have not discussed
the question of *completeness* for the inference algorithms that use them. Search algorithms
such as iterative deepening search (page 81) are complete in the sense that they will find
any reachable goal, but if the available inference rules are inadequate, then the goal is not
reachable—no proof exists that uses only those inference rules. For example, if we removed
the biconditional elimination rule, the proof in the preceding section would not go through.
The current section introduces a single inference rule, **resolution**, that yields a complete
inference algorithm when coupled with any complete search algorithm.

We begin by using a simple version of the resolution rule in the wumpus world. Let us
consider the steps leading up to Figure 7.4(a): the agent returns from [2,1] to [1,1] and then
goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the
knowledge base:

$R_{11}:$ $\neg B_{1,2}$.
$R_{12}:$ $B_{1,2} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{1,3})$.

By the same process that led to R_{10} earlier, we can now derive the absence of pits in [2,2] and
[1,3] (remember that [1,1] is already known to be pitless):

$R_{13}:$ $\neg P_{2,2}$.
$R_{14}:$ $\neg P_{1,3}$.

We can also apply biconditional elimination to R_3, followed by Modus Ponens with R_5, to
obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

$R_{15}:$ $P_{1,1} \vee P_{2,2} \vee P_{3,1}$.

Now comes the first application of the resolution rule: the literal $\neg P_{2,2}$ in R_{13} *resolves with*
the literal $P_{2,2}$ in R_{15} to give the **resolvent**

$R_{16}:$ $P_{1,1} \vee P_{3,1}$.

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1]
or [3,1]. Similarly, the literal $\neg P_{1,1}$ in R_1 resolves with the literal $P_{1,1}$ in R_{16} to give

$R_{17}:$ $P_{3,1}$.

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last
two inference steps are examples of the **unit resolution** inference rule

---

[9] **Nonmonotonic** logics, which violate the monotonicity property, capture a common property of human rea-
soning: changing one's mind. They are discussed in Section 10.6.

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k}$$

Complementary literals
Clause

where each ℓ is a literal and ℓ_i and m are **complementary literals** (i.e., one is the negation of the other). Thus, the unit resolution rule takes a **clause**—a disjunction of literals—and a literal and produces a new clause. Note that a single literal can be viewed as a disjunction of one literal, also known as a **unit clause**.

Unit clause

Resolution

The unit resolution rule can be generalized to the full **resolution** rule

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

where ℓ_i and m_j are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses *except* the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

You can resolve only one pair of complementary literals at a time. For example, we can resolve P and $\neg P$ to deduce

$$\frac{P \vee \neg Q \vee R, \qquad \neg P \vee Q}{\neg Q \vee Q \vee R},$$

but you can't resolve on both P and Q at once to infer R. There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.[10] The removal of multiple copies of literals is called **factoring**. For example, if we resolve $(A \vee B)$ with $(A \vee \neg B)$, we obtain $(A \vee A)$, which is reduced to just A by factoring.

Factoring

The *soundness* of the resolution rule can be seen easily by considering the literal ℓ_i that is complementary to literal m_j in the other clause. If ℓ_i is true, then m_j is false, and hence $m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n$ must be true, because $m_1 \vee \cdots \vee m_n$ is given. If ℓ_i is false, then $\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k$ must be true because $\ell_1 \vee \cdots \vee \ell_k$ is given. Now ℓ_i is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.

What is more surprising about the resolution rule is that it forms the basis for a family of *complete* inference procedures. *A resolution-based theorem prover can, for any sentences α and β in propositional logic, decide whether $\alpha \models \beta$.* The next two subsections explain how resolution accomplishes this.

Conjunctive normal form

The resolution rule applies only to clauses (that is, disjunctions of literals), so it would seem to be relevant only to knowledge bases and queries consisting of clauses. How, then, can it lead to a complete inference procedure for all of propositional logic? The answer is that *every sentence of propositional logic is logically equivalent to a conjunction of clauses.*

Conjunctive normal form
CNF

A sentence expressed as a conjunction of clauses is said to be in **conjunctive normal form** or **CNF** (see Figure 7.12). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$ into CNF. The steps are as follows:

---

[10] If a clause is viewed as a *set* of literals, then this restriction is automatically respected. Using set notation for clauses makes the resolution rule much cleaner, at the cost of introducing additional notation.

$$\begin{array}{rcl}
CNFSentence & \rightarrow & Clause_1 \wedge \cdots \wedge Clause_n \\
Clause & \rightarrow & Literal_1 \vee \cdots \vee Literal_m \\
Fact & \rightarrow & Symbol \\
Literal & \rightarrow & Symbol \mid \neg Symbol \\
Symbol & \rightarrow & P \mid Q \mid R \mid \ldots \\
HornClauseForm & \rightarrow & DefiniteClauseForm \mid GoalClauseForm \\
DefiniteClauseForm & \rightarrow & Fact \mid (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow Symbol \\
GoalClauseForm & \rightarrow & (Symbol_1 \wedge \cdots \wedge Symbol_l) \Rightarrow False
\end{array}$$

Figure 7.12 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A CNF clause such as $\neg A \vee \neg B \vee C$ can be written in definite clause form as $A \wedge B \Rightarrow C$.

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.
$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1}).$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1}).$$

3. CNF requires $\neg$ to appear only in literals, so we "move $\neg$ inwards" by repeated application of the following equivalences from Figure 7.11:

$$\begin{array}{ll}
\neg(\neg\alpha) \equiv \alpha & \text{(double-negation elimination)} \\
\neg(\alpha \wedge \beta) \equiv (\neg\alpha \vee \neg\beta) & \text{(De Morgan)} \\
\neg(\alpha \vee \beta) \equiv (\neg\alpha \wedge \neg\beta) & \text{(De Morgan)}
\end{array}$$

In the example, we require just one application of the last rule:
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1}).$$

4. Now we have a sentence containing nested $\wedge$ and $\vee$ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing $\vee$ over $\wedge$ wherever possible.
$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1}).$$

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A resolution algorithm

Inference procedures based on resolution work by using the principle of proof by contradiction introduced on page 223. That is, to show that $KB \models \alpha$, we show that $(KB \wedge \neg\alpha)$ is unsatisfiable. We do this by proving a contradiction.

A resolution algorithm is shown in Figure 7.13. First, $(KB \wedge \neg\alpha)$ is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present. The process continues until one of two things happens:

- there are no new clauses that can be added, in which case KB does not entail α; or,
- two clauses resolve to yield the *empty* clause, in which case KB entails α.

---

function PL-RESOLUTION(KB, α) **returns** *true* or *false*
    inputs: KB, the knowledge base, a sentence in propositional logic
           α, the query, a sentence in propositional logic

    clauses ← the set of clauses in the CNF representation of $KB \wedge \neg\alpha$
    new ← { }
    while *true* **do**
        for each pair of clauses C_i, C_j **in** *clauses* **do**
            resolvents ← PL-RESOLVE(C_i, C_j)
            if *resolvents* contains the empty clause **then return** *true*
            new ← *new* ∪ *resolvents*
        if *new* ⊆ *clauses* **then return** *false*
        clauses ← *clauses* ∪ *new*

Figure 7.13 A simple resolution algorithm for propositional logic. PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

The empty clause—a disjunction of no disjuncts—is equivalent to *False* because a disjunction is true only if at least one of its disjuncts is true. Moreover, the empty clause arises only from resolving two contradictory unit clauses such as P and $\neg P$.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

$$KB = R_2 \wedge R_4 = (B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})) \wedge \neg B_{1,1}$$

and we wish to prove α, which is, say, $\neg P_{1,2}$. When we convert $(KB \wedge \neg\alpha)$ into CNF, we obtain the clauses shown at the top of Figure 7.14. The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when $P_{1,2}$ is resolved with $\neg P_{1,2}$, we obtain the empty clause, shown as a small square. Inspection of Figure 7.14 reveals that many resolution steps are pointless. For example, the clause $B_{1,1} \vee \neg B_{1,1} \vee P_{1,2}$ is equivalent to *True* $\vee P_{1,2}$ which is equivalent to *True*. Deducing that *True* is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded.

Completeness of resolution

To conclude our discussion of resolution, we now show why PL-RESOLUTION is complete. To do this, we introduce the **resolution closure** $RC(S)$ of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable *clauses*. It is easy to see that $RC(S)$ must be finite: thanks to the factoring step, there are only finitely many distinct clauses that can be constructed out of the symbols $P_1, \ldots, P_k$ that appear in S. Hence, PL-RESOLUTION always terminates.

The completeness theorem for resolution in propositional logic is called the **ground resolution theorem**:

> If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

Resolution closure
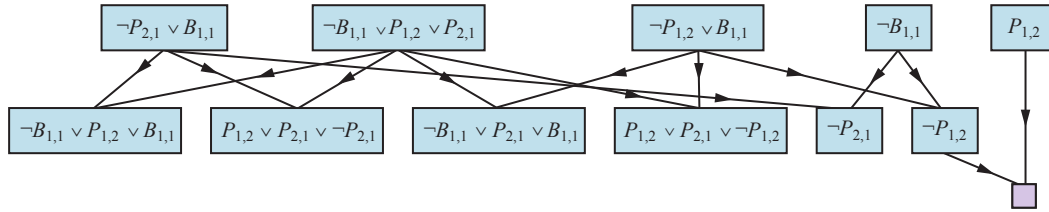
Ground resolution theorem

Figure 7.14 Partial application of PL-RESOLUTION to a simple inference in the wumpus world to prove the query $\neg P_{1,2}$. Each of the leftmost four clauses in the top row is paired with each of the other three, and the resolution rule is applied to yield the clauses on the bottom row. We see that the third and fourth clauses on the top row combine to yield the clause $\neg P_{1,2}$, which is then resolved with $P_{1,2}$ to yield the empty clause, meaning that the query is proven.

This theorem is proved by demonstrating its contrapositive: if the closure $RC(S)$ does *not* contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for $P_1, \ldots, P_k$. The construction procedure is as follows:

For i from 1 to k,

– If a clause in $RC(S)$ contains the literal $\neg P_i$ and all its other literals are false under the assignment chosen for $P_1, \ldots, P_{i-1}$, then assign *false* to P_i.
– Otherwise, assign *true* to P_i.

This assignment to $P_1, \ldots, P_k$ is a model of S. To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol P_i causes some clause C to become false. For this to happen, it must be the case that all the *other* literals in C must already have been falsified by assignments to $P_1, \ldots, P_{i-1}$. Thus, C must now look like either $(false \lor false \lor \cdots false \lor P_i)$ or like $(false \lor false \lor \cdots false \lor \neg P_i)$. If just one of these two is in $RC(S)$, then the algorithm will assign the appropriate truth value to P_i to make C true, so C can only be falsified if *both* of these clauses are in $RC(S)$.

Now, since $RC(S)$ is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to $P_1, \ldots, P_{i-1}$. This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in $RC(S)$; that is, it produces a model of $RC(S)$. Finally, because S is contained in $RC(S)$, any model of $RC(S)$ is a model of S itself.

7.5.3 Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed. Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

One such restricted form is the **definite clause**, which is a disjunction of literals of which *exactly one is positive*. For example, the clause $(\neg L_{1,1} \lor \neg Breeze \lor B_{1,1})$ is a definite clause, whereas $(\neg B_{1,1} \lor P_{1,2} \lor P_{2,1})$ is not, because it has two positive clauses.

Slightly more general is the **Horn clause**, which is a disjunction of literals of which *at*

Definite clause

Horn clause

most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called **goal clauses**. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause. One more class is the *k*-CNF sentence, which is a CNF sentence where each clause has at most *k* literals.

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal. (See Exercise 7.DISJ.) For example, the definite clause $(\neg L_{1,1} \vee \neg Breeze \vee B_{1,1})$ can be written as the implication $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$. In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze percept, then [1,1] is breezy. In Horn form, the premise is called the **body** and the conclusion is called the **head**. A sentence consisting of a single positive literal, such as $L_{1,1}$, is called a **fact**. It too can be written in implication form as $True \Rightarrow L_{1,1}$, but it is simpler to write just $L_{1,1}$.

2. Inference with Horn clauses can be done through the **forward-chaining** and **backward-chaining** algorithms, which we explain next. Both of these algorithms are natural, in that the inference steps are obvious and easy for humans to follow. This type of inference is the basis for **logic programming**, which is discussed in Chapter 9.

3. Deciding entailment with Horn clauses can be done in time that is *linear* in the size of the knowledge base—a pleasant surprise.

7.5.4 Forward and backward chaining

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses. It begins from known facts (positive literals) in the knowledge base. If all the premises of an implication are known, then its conclusion is added to the set of known facts. For example, if $L_{1,1}$ and *Breeze* are known and $(L_{1,1} \wedge Breeze) \Rightarrow B_{1,1}$ is in the knowledge base, then $B_{1,1}$ can be added. This process continues until the query q is added or until no further inferences can be made. The algorithm is shown in Figure 7.15; the main point to remember is that it runs in linear time.

The best way to understand the algorithm is through an example and a picture. Figure 7.16(a) shows a simple knowledge base of Horn clauses with A and B as known facts. Figure 7.16(b) shows the same knowledge base drawn as an **AND–OR graph** (see Chapter 4). In AND–OR graphs, multiple edges joined by an arc indicate a conjunction—every edge must be proved—while multiple edges without an arc indicate a disjunction—any edge can be proved. It is easy to see how forward chaining works in the graph. The known leaves (here, A and B) are set, and inference propagates up the graph as far as possible. Wherever a conjunction appears, the propagation waits until all the conjuncts are known before proceeding. The reader is encouraged to work through the example in detail.

It is easy to see that forward chaining is **sound**: every inference is essentially an application of Modus Ponens. Forward chaining is also **complete**: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the *inferred* table (after the algorithm reaches a fixed point where no new inferences are possible). The table contains *true* for each symbol inferred during the process, and *false* for all other symbols. We can view the table as a logical model; moreover, *every definite clause in the original KB is true in this model*.

---

function PL-FC-ENTAILS?(*KB*, *q*) **returns** *true* or *false*
    inputs: *KB*, the knowledge base, a set of propositional definite clauses
            q, the query, a proposition symbol
    count ← a table, where *count*[*c*] is initially the number of symbols in clause *c*'s premise
    inferred ← a table, where *inferred*[*s*] is initially *false* for all symbols
    queue ← a queue of symbols, initially symbols known to be true in *KB*

    while *queue* is not empty **do**
        p ← POP(*queue*)
        if *p* = *q* **then return** *true*
        if *inferred*[*p*] = *false* **then**
            inferred[*p*] ← *true*
            for each clause *c* in *KB* where *p* is in *c*.PREMISE **do**
                decrement *count*[*c*]
                if *count*[*c*] = 0 **then** add *c*.CONCLUSION to *queue*
    return *false*

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are not yet proven. Whenever a new symbol *p* from the agenda is processed, the count is reduced by one for each implication in whose premise *p* appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

To see this, assume the opposite, namely that some clause $a_1 \wedge \ldots \wedge a_k \Rightarrow b$ is false in the model. Then $a_1 \wedge \ldots \wedge a_k$ must be true in the model and *b* must be false in the model. But this contradicts our assumption that the algorithm has reached a fixed point, because we would now be licensed to add *b* to the KB. We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB. Furthermore, any atomic sentence *q* that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence *q* must be inferred by the algorithm.

Forward chaining is an example of the general concept of **data-driven** reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind. For example, the wumpus agent might TELL its percepts to the knowledge base using an incremental forward-chaining algorithm in which new facts can be added to the agenda to initiate new inferences. In humans, a certain amount of data-driven reasoning occurs as new information arrives. For example, if I am indoors and hear rain starting to fall, it might occur to me that the picnic will be canceled. Yet it will probably not occur to me that the seventeenth petal on the largest rose in my neighbor's garden will get wet; humans keep forward chaining under careful control, lest they be swamped with irrelevant consequences.

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query *q* is known to be true, then no work is needed. Otherwise, the algorithm finds
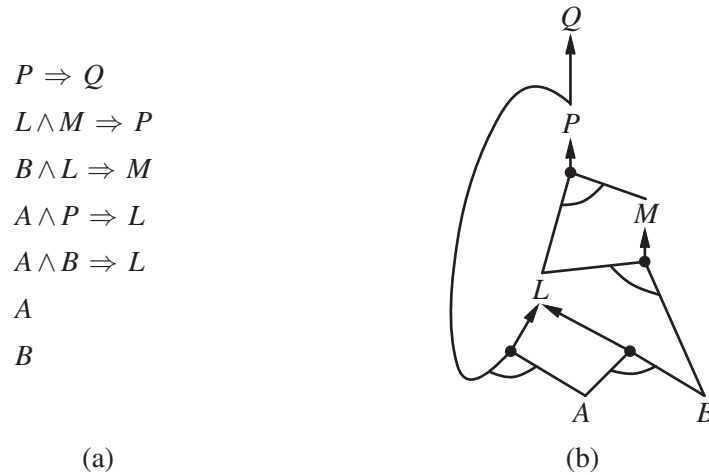
Data-driven

$$P \Rightarrow Q$$
$$L \wedge M \Rightarrow P$$
$$B \wedge L \Rightarrow M$$
$$A \wedge P \Rightarrow L$$
$$A \wedge B \Rightarrow L$$
$$A$$
$$B$$

(a)   (b)

Figure 7.16 (a) A set of Horn clauses. (b) The corresponding AND–OR graph.

those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 7.16, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof. The algorithm is essentially identical to the AND-OR-GRAPH-SEARCH algorithm in Figure 4.11. As with forward chaining, an efficient implementation runs in linear time.

Backward chaining is a form of **goal-directed reasoning**. It is useful for answering specific questions such as "What shall I do now?" and "Where are my keys?" Often, the cost of backward chaining is *much less* than linear in the size of the knowledge base, because the process touches only relevant facts.

Goal-directed
reasoning

7.6 Effective Propositional Model Checking

In this section, we describe two families of efficient algorithms for general propositional inference based on model checking: one approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the "technology" of propositional logic. This section can be skimmed on a first reading of the chapter.

The algorithms we describe are for checking satisfiability: the SAT problem. (As noted in Section 7.5, testing entailment, $\alpha \models \beta$, can be done by testing *un*satisfiability of $\alpha \wedge \neg\beta$.) We mentioned on page 223 the connection between finding a satisfying model for a logical sentence and finding a solution for a constraint satisfaction problem, so it is perhaps not surprising that the two families of propositional satisfiability algorithms closely resemble the backtracking algorithms of Section 6.3 and the local search algorithms of Section 6.4. They are, however, extremely important in their own right because so many combinatorial problems in computer science can be reduced to checking the satisfiability of a propositional sentence. Any improvement in satisfiability algorithms has huge consequences for our ability to handle complexity in general.

7.6.1 A complete backtracking algorithm

The first algorithm we consider is often called the **Davis–Putnam algorithm**, after the sem- Davis–Putnam algorithm
inal paper by Martin Davis and Hilary Putnam (1960). The algorithm is in fact the version
described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the ini-
tials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set
of clauses. Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive,
depth-first enumeration of possible models. It embodies three improvements over the simple
scheme of TT-ENTAILS?:

- *Early termination*: The algorithm detects whether the sentence must be true or false,
  even with a partially completed model. A clause is true if *any* literal is true, even if
  the other literals do not yet have truth values; hence, the sentence as a whole could be
  judged true even before the model is complete. For example, the sentence $(A \vee B) \wedge$
  $(A \vee C)$ is true if A is true, regardless of the values of B and C. Similarly, a sentence is
  false if *any* clause is false, which occurs when each of its literals is false. Again, this
  can occur long before the model is complete. Early termination avoids examination of
  entire subtrees in the search space.

- *Pure symbol heuristic*: A **pure symbol** is a symbol that always appears with the same Pure symbol
  "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$,
  the symbol A is pure because only the positive literal appears, B is pure because only the
  negative literal appears, and C is impure. It is easy to see that if a sentence has a model,
  then it has a model with the pure symbols assigned so as to make their literals *true*,
  because doing so can never make a clause false. Note that, in determining the purity
  of a symbol, the algorithm can ignore clauses that are already known to be true in the
  model constructed so far. For example, if the model contains $B = false$, then the clause
  $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive
  literal; therefore C becomes pure.

- *Unit clause heuristic*: A **unit clause** was defined earlier as a clause with just one literal.
  In the context of DPLL, it also means clauses in which all literals but one are already
  assigned *false* by the model. For example, if the model contains $B = true$, then $(\neg B \vee$
  $\neg C)$ simplifies to $\neg C$, which is a unit clause. Obviously, for this clause to be true, C must
  be set to *false*. The unit clause heuristic assigns all such symbols before branching on
  the remainder. One important consequence of the heuristic is that any attempt to prove
  (by refutation) a literal that is already in the knowledge base will succeed immediately
  (Exercise 7.KNOW). Notice also that assigning one unit clause can create another unit
  clause—for example, when C is set to *false*, $(C \vee A)$ becomes a unit clause, causing *true*
  to be assigned to A. This "cascade" of forced assignments is called **unit propagation**. Unit propagation
  It resembles the process of forward chaining with definite clauses, and indeed, if the
  CNF expression contains only definite clauses then DPLL essentially replicates forward
  chaining. (See Exercise 7.DPLL.)

The DPLL algorithm is shown in Figure 7.17, which gives the essential skeleton of the search
process without the implementation details.

What Figure 7.17 does not show are the tricks that enable SAT solvers to scale up to large
problems. It is interesting that most of these tricks are in fact rather general, and we have
seen them before in other guises:

---

function DPLL-SATISFIABLE?(*s*) **returns** *true* or *false*
  inputs: *s*, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of *s*
  symbols ← a list of the proposition symbols in *s*
  return DPLL(*clauses*, *symbols*, { })

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

  if every clause in *clauses* is true in *model* **then return** *true*
  if some clause in *clauses* is false in *model* **then return** *false*
  P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
  if *P* is non-null **then return** DPLL(*clauses*, *symbols* – *P*, *model* ∪ {*P=value*})
  P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
  if *P* is non-null **then return** DPLL(*clauses*, *symbols* – *P*, *model* ∪ {*P=value*})
  P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
  return DPLL(*clauses*, *rest*, *model* ∪ {*P=true*}) **or**
        DPLL(*clauses*, *rest*, *model* ∪ {*P=false*}))

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

---

1. **Component analysis** (as seen with Tasmania in CSPs): As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called **components**, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.

2. **Variable and value ordering** (as seen in Section 6.3.1 for CSPs): Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value *true* before *false*. The **degree heuristic** (see page 193) suggests choosing the variable that appears most frequently over all remaining clauses.

3. **Intelligent backtracking** (as seen in Section 6.3.3 for CSPs): Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict. All SAT solvers that do intelligent backtracking use some form of **conflict clause learning** to record conflicts so that they won't be repeated later in the search. Usually a limited-size set of conflicts is kept, and rarely used ones are dropped.

4. **Random restarts** (as seen on page 113 for hill climbing): Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices (in variable and value selection) are made. Clauses that are learned in the first run are retained after the restart and can help prune the search space. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.

5. **Clever indexing** (as seen in many algorithms): The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things

---

function WALKSAT(*clauses*, *p*, *max_flips*) **returns** a satisfying model or *failure*
   inputs: *clauses*, a set of clauses in propositional logic
        p, the probability of choosing to do a "random walk" move, typically around 0.5
        max_flips, number of value flips allowed before giving up

   model ← a random assignment of *true*/*false* to the symbols in *clauses*
   for each *i* = 1 **to** *max_flips* **do**
      if *model* satisfies *clauses* **then return** *model*
      clause ← a randomly selected clause from *clauses* that is false in *model*
      if RANDOM(0, 1) ≤ *p* **then**
         flip the value in *model* of a randomly selected symbol from *clause*
      else flip whichever symbol in *clause* maximizes the number of satisfied clauses
   return *failure*

Figure 7.18 The WALKSAT algorithm for checking satisfiability by randomly flipping the values of variables. Many versions of the algorithm exist.

---

as "the set of clauses in which variable X_i appears as a positive literal." This task is complicated by the fact that the algorithms are interested only in the clauses that have not yet been satisfied by previous assignments to variables, so the indexing structures must be updated dynamically as the computation proceeds.

With these enhancements, modern solvers can handle problems with tens of millions of variables. They have revolutionized areas such as hardware verification and security protocol verification, which previously required laboriou, hand-guided proofs.

7.6.2 Local search algorithms

We have seen several local search algorithms so far in this book, including HILL-CLIMBING (page 111) and SIMULATED-ANNEALING (page 115). These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function. Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job. In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs (page 198). All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time. The space usually contains many local minima, to escape from which various forms of randomness are required. In recent years, there has been a great deal of experimentation to find a good balance between greediness and randomness.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 7.18). On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip. It chooses randomly between two ways to pick which symbol to flip: (1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state and (2) a "random walk" step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns *failure*, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time. If we set *max_flips* = ∞ and *p* > 0, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit upon the

solution. Alas, if *max_flips* is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

For this reason, WALKSAT is most useful when we expect a solution to exist—for example, the problems discussed in Chapters 3 and 6 usually have solutions. On the other hand, WALKSAT cannot always detect *unsatisfiability*, which is required for deciding entailment. For example, an agent cannot *reliably* use WALKSAT to prove that a square is safe in the wumpus world. Instead, it can say, "I thought about it for an hour and couldn't come up with a possible world in which the square *isn't* safe." This may be a good empirical indicator that the square is safe, but it's certainly not a proof.

7.6.3 The landscape of random SAT problems

Some SAT problems are harder than others. *Easy* problems can be solved by any old algorithm, but because we know that SAT is NP-complete, at least some problem instances must require exponential run time. In Chapter 6, we saw some surprising discoveries about certain kinds of problems. For example, the *n*-queens problem—thought to be quite tricky for backtracking search algorithms—turned out to be trivially easy for local search methods, such as min-conflicts. This is because solutions are very densely distributed in the space of assignments, and any initial assignment is guaranteed to have a solution nearby. Thus, *n*-queens is

Underconstrained   easy because it is **underconstrained**.

When we look at satisfiability problems in conjunctive normal form, an underconstrained problem is one with relatively *few* clauses constraining the variables. For example, here is a randomly generated 3-CNF sentence with five symbols and five clauses:

$$(\neg D \vee \neg B \vee C) \wedge (B \vee \neg A \vee \neg C) \wedge (\neg C \vee \neg B \vee E)$$
$$\wedge (E \vee \neg D \vee B) \wedge (B \vee E \vee \neg C).$$

Sixteen of the 32 possible assignments are models of this sentence, so, on average, it would take just two random guesses to find a model. This is an easy satisfiability problem, as are most such underconstrained problems. On the other hand, an *overconstrained* problem has many clauses relative to the number of variables and is likely to have no solutions. Overconstrained problems are often easy to solve, because the constraints quickly lead either to a solution or to a dead end from which there is no escape.

To go beyond these basic intuitions, we must define exactly how random sentences are generated. The notation $CNF_k(m,n)$ denotes a k-CNF sentence with m clauses and n symbols, where the clauses are chosen uniformly, independently, and without replacement from among all clauses with k different literals, which are positive or negative at random. (A symbol may not appear twice in a clause, nor may a clause appear twice in a sentence.)

Given a source of random sentences, we can measure the probability of satisfiability. Figure 7.19(a) plots the probability for $CNF_3(m,50)$, that is, sentences with 50 variables and 3 literals per clause, as a function of the clause/symbol ratio, m/n. As we expect, for small m/n the probability of satisfiability is close to 1, and at large m/n the probability is close to 0. The probability drops fairly sharply around $m/n = 4.3$. Empirically, we find that the "cliff" stays in roughly the same place (for $k = 3$) and gets sharper and sharper as n increases.

Satisfiability
threshold conjecture   Theoretically, the **satisfiability threshold conjecture** says that for every $k \geq 3$, there is a threshold ratio r_k such that, as n goes to infinity, the probability that $CNF_k(rn,n)$ is satisfiable becomes 1 for all values of r below the threshold, and 0 for all values above. The conjecture remains unproven, even for special cases like $k = 3$. Whether it is a theorem or not, this kind
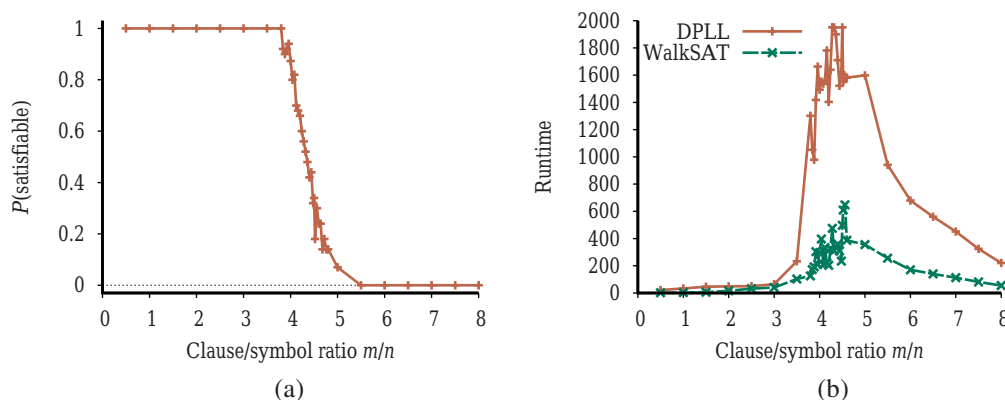
Figure 7.19 (a) Graph showing the probability that a random 3-CNF sentence with $n = 50$ symbols is satisfiable, as a function of the clause/symbol ratio m/n. (b) Graph of the median run time (measured in number of iterations) for both DPLL and WALKSAT on random 3-CNF sentences. The most difficult problems have a clause/symbol ratio of about 4.3.

of thresholding effect is certainly common, for satisfiability problems as well as other types of NP-hard problems.

Now that we have a good idea where the satisfiable and unsatisfiable problems are, the next question is, where are the hard problems? It turns out that they are also often at the threshold value. Figure 7.19(b) shows that 50-symbol problems at the threshold value of 4.3 are about 20 times more difficult to solve than those at a ratio of 3.3. The underconstrained problems are easiest to solve (because it is so easy to guess a solution); the overconstrained problems are not as easy as the underconstrained, but still are much easier than the ones right at the threshold.

7.7 Agents Based on Propositional Logic

In this section, we bring together what we have learned so far in order to construct wumpus world agents that use propositional logic. The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history. This requires writing down a complete logical model of the effects of actions. We then show how logical inference can be used by an agent in the wumpus world. We also show how the agent can keep track of the world efficiently without going back into the percept history for each inference. Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals, provided its knowledge base is true in the actual world.

7.7.1 The current state of the world

As stated at the beginning of the chapter, a logical agent operates by deducing what to do from a knowledge base of sentences about the world. The knowledge base is composed of axioms—general knowledge about how the world works—and percept sentences obtained from the agent's experience in a particular world. In this section, we focus on the problem of deducing the current state of the wumpus world—where am I, is that square safe, and so on.

We began collecting axioms in Section 7.4.3. The agent knows that the starting square contains no pit ($\neg P_{1,1}$) and no wumpus ($\neg W_{1,1}$). Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus. Thus, we include a large collection of sentences of the following form:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\cdots$$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is *at least one* wumpus:

$$W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,3} \vee W_{4,4}\,.$$

Then we have to say that there is *at most one* wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\neg W_{1,1} \vee \neg W_{1,2}$$
$$\neg W_{1,1} \vee \neg W_{1,3}$$
$$\cdots$$
$$\neg W_{4,3} \vee \neg W_{4,4}\,.$$

So far, so good. Now let's consider the agent's percepts. We are using $S_{1,1}$ to mean there is a stench in [1,1]; can we use a single proposition, *Stench* to mean that the agent perceives a stench? Unfortunately we can't: if there was no stench at the previous time step, then $\neg Stench$ would already be asserted, and the new assertion would simply result in a contradiction. The problem is solved when we realize that a percept asserts something *only about the current time*. Thus, if the time step (as supplied to MAKE-PERCEPT-SENTENCE in Figure 7.1) is 4, then we add $Stench^4$ to the knowledge base, rather than *Stench*—neatly avoiding any contradiction with $\neg Stench^3$. The same goes for the breeze, bump, glitter, and scream percepts.

The idea of associating propositions with time steps extends to any aspect of the world that changes over time. For example, the initial knowledge base includes $L^0_{1,1}$—the agent is in square [1, 1] at time 0—as well as $FacingEast^0$, $HaveArrow^0$, and $WumpusAlive^0$. We use the noun **fluent** (from the Latin *fluens*, flowing) to refer to an aspect of the world that changes. "Fluent" is a synonym for "state variable," in the sense described in the discussion of factored representations in Section 2.4.7 on page 58. Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called **atemporal variables**.

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced as follows.[11] For any time step t and any square $[x, y]$, we assert

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y})\,.$$

Now, of course, we need axioms that allow the agent to keep track of fluents such as $L^t_{x,y}$. These fluents change as the result of actions taken by the agent, so, in the terminology of Chapter 3, we need to write down the **transition model** of the wumpus world as a set of logical sentences.

Fluent

Atemporal variable

---

[11] Section 7.4.3 conveniently glossed over this requirement.

First we need proposition symbols for the occurrences of actions. As with percepts, these symbols are indexed by time; thus, $Forward^0$ means that the agent executes the *Forward* action at time 0. By convention, the percept for a given time step happens first, followed by the action for that time step, followed by a transition to the next time step.

To describe how the world changes, we can try writing **effect axioms** that specify the Effect axiom outcome of an action at the next time step. For example, if the agent is at location $[1,1]$ facing east at time 0 and goes *Forward*, the result is that the agent is in square $[2,1]$ and no longer is in $[1,1]$:

$$L_{1,1}^0 \wedge FacingEast^0 \wedge Forward^0 \;\Rightarrow\; (L_{2,1}^1 \wedge \neg L_{1,1}^1). \tag{7.1}$$

We would need one such sentence for each possible time step, for each of the 16 squares, and each of the four orientations. We would also need similar sentences for the other actions: *Grab*, *Shoot*, *Climb*, *TurnLeft*, and *TurnRight*.

Let us suppose that the agent does decide to move *Forward* at time 0 and asserts this fact into its knowledge base. Given the effect axiom in Equation (7.1), combined with the initial assertions about the state at time 0, the agent can now deduce that it is in $[2,1]$. That is, $\text{ASK}(KB, L_{2,1}^1) = true$. So far, so good. Unfortunately, if we $\text{ASK}(KB, HaveArrow^1)$, the answer is *false*, that is, the agent cannot prove it still has the arrow; nor can it prove it *doesn't* have it! The information has been lost because the effect axiom fails to state what remains *unchanged* as the result of an action. The need to do this gives rise to the **frame problem**.[12] Frame problem One possible solution to the frame problem would be to add **frame axioms** explicitly asserting Frame axiom all the propositions that remain the same. For example, for each time t we would have

$$Forward^t \;\Rightarrow\; (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \;\Rightarrow\; (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$
$$\dots$$

where we explicitly mention every proposition that stays unchanged from time t to time $t+1$ under the action *Forward*. Although the agent now knows that it still has the arrow after moving forward and that the wumpus hasn't died or come back to life, the proliferation of frame axioms seems remarkably inefficient. In a world with m different actions and n fluents, the set of frame axioms will be of size $O(mn)$. This specific manifestation of the frame problem is sometimes called the **representational frame problem**. The problem played a Representational frame problem significant role in the history of AI; we explore it further in the notes at the end of the chapter.

The representational frame problem is significant because the real world has very many fluents, to put it mildly. Fortunately for us humans, each action typically changes no more than some small number k of those fluents—the world exhibits **locality**. Solving the repre- Locality sentational frame problem requires defining the transition model with a set of axioms of size $O(mk)$ rather than size $O(mn)$. There is also an **inferential frame problem**: the problem of Inferential frame problem projecting forward the results of a t-step plan of action in time $O(kt)$ rather than $O(nt)$.

The solution to the problem involves changing one's focus from writing axioms about *actions* to writing axioms about *fluents*. Thus for each fluent F, we will have an axiom that defines the truth value of F^{t+1} in terms of fluents (including F itself) at time t and the actions that may have occurred at time t. Now, the truth value of F^{t+1} can be set in one of two ways:

---

[12] The name "frame problem" comes from "frame of reference" in physics—the assumed stationary background with respect to which motion is measured. It also has an analogy to the frames of a movie, in which normally most of the background stays constant while changes occur in the foreground.

either the action at time t causes F to be true at $t + 1$, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a **successor-state axiom** and has this form:

Successor-state axiom

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t) .$$

One of the simplest successor-state axioms is the one for *HaveArrow*. Because there is no action for reloading, the $ActionCausesF^t$ part goes away and we are left with

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) . \tag{7.2}$$

For the agent's location, the successor-state axioms are more elaborate. For example, $L_{1,1}^{t+1}$ is true if either (a) the agent moved *Forward* from $[1,2]$ when facing south, or from $[2,1]$ when facing west; or (b) $L_{1,1}^t$ was already true and the action did not cause movement (either because the action was not *Forward* or because the action bumped into a wall). Written out in propositional logic, this becomes

$$
\begin{aligned}
L_{1,1}^{t+1} \quad \Leftrightarrow \quad & (L_{1,1}^t \wedge (\neg Forward^t \vee Bump^{t+1})) \\
\vee \quad & (L_{1,2}^t \wedge (FacingSouth^t \wedge Forward^t)) \\
\vee \quad & (L_{2,1}^t \wedge (FacingWest^t \wedge Forward^t)) .
\end{aligned} \tag{7.3}
$$

Exercise 7.SSAX asks you to write out axioms for the remaining wumpus world fluents.

Given a complete set of successor-state axioms and the other axioms listed at the beginning of this section, the agent will be able to ASK and answer any answerable question about the current state of the world. For example, in Section 7.2 the initial sequence of percepts and actions is

$$\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \ ; \ Forward^0$$
$$\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \ ; \ TurnRight^1$$
$$\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \ ; \ TurnRight^2$$
$$\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \ ; \ Forward^3$$
$$\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \ ; \ TurnRight^4$$
$$\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \ ; \ Forward^5$$
$$Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6$$

At this point, we have $\text{ASK}(KB, L_{1,2}^6) = true$, so the agent knows where it is. Moreover, $\text{ASK}(KB, W_{1,3}) = true$ and $\text{ASK}(KB, P_{3,1}) = true$, so the agent has found the wumpus and one of the pits. The most important question for the agent is whether a square is OK to move into—that is, whether the square is free of a pit or live wumpus. It's convenient to add axioms for this, having the form

$$OK_{x,y}^t \Leftrightarrow \neg P_{x,y} \wedge \neg (W_{x,y} \wedge WumpusAlive^t) .$$

Finally, $\text{ASK}(KB, OK_{2,2}^6) = true$, so the square $[2,2]$ is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK—and can do so in just a few milliseconds for small-to-medium wumpus worlds.

Solving the representational and inferential frame problems is a big step forward, but a pernicious problem remains: we need to confirm that *all* the necessary preconditions of an action hold for it to have its intended effect. We said that the *Forward* action moves the agent

ahead unless there is a wall in the way, but there are many other unusual exceptions that could cause the action to fail: the agent might trip and fall, be stricken with a heart attack, be carried away by giant bats, etc. Specifying all these exceptions is called the **qualification problem**. There is no complete solution within logic; system designers have to use good judgment in deciding how detailed they want to be in specifying their model, and what details they want to leave out. We will see in Chapter 12 that probability theory allows us to summarize all the exceptions without explicitly naming them.

Qualification problem

7.7.2 A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straight-forwardly with condition–action rules (see Section 2.4.2) and with problem-solving algorithms from Chapters 3 and 4 to produce a **hybrid agent** for the wumpus world. Figure 7.20 shows one possible way to do this. The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the *atemporal* axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits. At each time step, the new percept sentence is added along with all the axioms that depend on t, such as the successor-state axioms. (The next section explains why the agent doesn't need axioms for *future* time steps.) Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

Hybrid agent

The main body of the agent program constructs a plan based on a decreasing priority of goals. First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave. Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares.

Route planning is done with A* search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where $\text{ASK}(KB, \neg W_{x,y})$ is false—that is, where it is *not* known that there is *not* a wumpus. The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot. If this fails, the program looks for a square to explore that is not provably unsafe—that is, a square for which $\text{ASK}(KB, \neg OK^t_{x,y})$ returns false. If there is no such square, then the mission is impossible and the agent retreats to $[1, 1]$ and climbs out of the cave.

7.7.3 Logical state estimation

The agent program in Figure 7.20 works quite well, but it has one major weakness: as time goes by, the computational expense involved in the calls to ASK goes up and up. This happens mainly because the required inferences have to go back further and further in time and involve more and more proposition symbols. Obviously, this is unsustainable—we cannot have an agent whose time to process each percept grows in proportion to the length of its life! What we really need is a *constant* update time—that is, independent of t. The obvious answer is to save, or **cache**, the results of inference, so that the inference process at the next time step can build on the results of earlier steps instead of having to start again from scratch.

Caching

As we saw in Section 4.4, the history of percepts and all their ramifications can be replaced by the **belief state**—that is, some representation of the set of all possible current states

---

function HYBRID-WUMPUS-AGENT(*percept*) **returns** an *action*
  inputs: *percept*, a list, [*stench,breeze,glitter,bump,scream*]
  persistent: *KB*, a knowledge base, initially the atemporal "wumpus physics"
          t, a counter, initially 0, indicating time
          plan, an action sequence, initially empty

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept, t*))
  TELL the *KB* the temporal "physics" sentences for time *t*
  safe ← {[*x, y*] : ASK(*KB, OK$_{x,y}^t$*) = *true*}
  if ASK(*KB, Glitter^t*) = *true* **then**
    plan ← [*Grab*] + PLAN-ROUTE(*current*, {[1,1]}, *safe*) + [*Climb*]
  if *plan* is empty **then**
    unvisited ← {[*x, y*] : ASK(*KB, L$_{x,y}^{t'}$*) = *false* for all *t'* ≤ *t*}
    plan ← PLAN-ROUTE(*current, unvisited* ∩ *safe, safe*)
  if *plan* is empty and ASK(*KB, HaveArrow^t*) = *true* **then**
    possible_wumpus ← {[*x, y*] : ASK(*KB,* ¬ *W$_{x,y}$*) = *false*}
    plan ← PLAN-SHOT(*current, possible_wumpus, safe*)
  if *plan* is empty **then**   // *no choice but to take a risk*
    not_unsafe ← {[*x, y*] : ASK(*KB,* ¬ *OK$_{x,y}^t$*) = *false*}
    plan ← PLAN-ROUTE(*current, unvisited* ∩ *not_unsafe, safe*)
  if *plan* is empty **then**
    plan ← PLAN-ROUTE(*current*, {[1,1]}, *safe*) + [*Climb*]
  action ← POP(*plan*)
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action, t*))
  t ← *t* + 1
  return *action*

function PLAN-ROUTE(*current,goals,allowed*) **returns** an action sequence
  inputs: *current*, the agent's current position
        goals, a set of squares; try to plan a route to one of them
        allowed, a set of squares that can form part of the route

  problem ← ROUTE-PROBLEM(*current, goals,allowed*)
  return SEARCH(*problem*)   // *Any search algorithm from Chapter 3*

Figure 7.20 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to choose actions. Each time HYBRID-WUMPUS-AGENT is called, it adds the percept to the knowledge base, and then either relies on a previously-defined plan or creates a new plan, and pops off the first step of the plan as the action to do next.

---

of the world.[13] The process of updating the belief state as new percepts arrive is called **state estimation** (see page 132). Whereas in Section 4.4 the belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the atemporal symbols. For example, the logical sentence

$$WumpusAlive^1 \wedge L_{2,1}^1 \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2}) \tag{7.4}$$

---

[13] We can think of the percept history itself as a representation of the belief state, but one that makes inference increasingly expensive as the history gets longer.
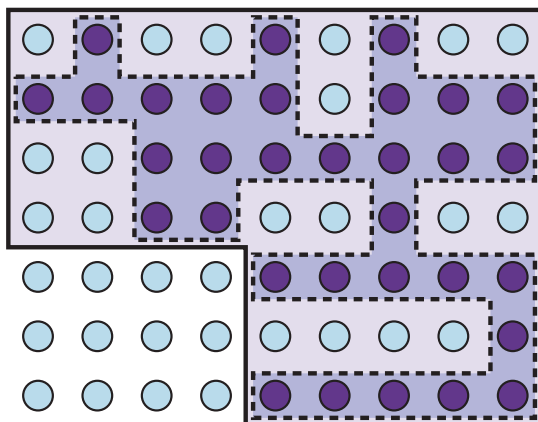
Figure 7.21 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

represents the set of all states at time 1 in which the wumpus is alive, the agent is at $[2,1]$, that square is breezy, and there is a pit in $[3,1]$ or $[2,2]$ or both.

Maintaining an exact belief state as a logical formula turns out not to be easy. If there are n fluent symbols for time t, then there are 2^n possible states—that is, assignments of truth values to those symbols. Now, the set of belief states is the powerset (set of all subsets) of the set of physical states. There are 2^n physical states, hence 2^{2^n} belief states. Even if we used the most compact possible encoding of logical formulas, with each belief state represented by a unique binary number, we would need numbers with $\log_2(2^{2^n}) = 2^n$ bits to label the current belief state. That is, exact state estimation may require logical formulas whose size is exponential in the number of symbols.

One very common and natural scheme for *approximate* state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove X^t and $\neg X^t$ for each symbol X^t (as well as each atemporal symbol whose truth value is not yet known), given the belief state at $t-1$. The conjunction of provable literals becomes the new belief state, and the previous belief state is discarded.

It is important to understand that this scheme may lose some information as time goes along. For example, if the sentence in Equation (7.4) were the true belief state, then neither $P_{3,1}$ nor $P_{2,2}$ would be provable individually and neither would appear in the 1-CNF belief state. (Exercise 7.HYBR explores one possible solution to this problem.) On the other hand, because every literal in the 1-CNF belief state is proved from the previous belief state, and the initial belief state is a true assertion, we know that the entire 1-CNF belief state must be true. Thus *the set of possible states represented by the 1-CNF belief state includes all states that are in fact possible given the full percept history.* As illustrated in Figure 7.21, the 1-CNF belief state acts as a simple outer envelope, or **conservative approximation**, around the exact belief state. We see this idea of conservative approximations to complicated sets as a recurring theme in many areas of AI.

◄

Conservative approximation

---

function SATPLAN(*init*, *transition*, *goal*, *T*_{max}) **returns** solution or *failure*
   inputs: *init*, *transition*, *goal*, constitute a description of the problem
         T_{max}, an upper limit for plan length

  for $t = 0$ **to** T_{max} **do**
    cnf ← TRANSLATE-TO-SAT(*init*, *transition*, *goal*, *t*)
    model ← SAT-SOLVER(*cnf*)
    if *model* is not null **then**
      return EXTRACT-SOLUTION(*model*)
  return *failure*

Figure 7.22 The SATPLAN algorithm. The planning problem is translated into a CNF sentence in which the goal is asserted to hold at a fixed time step *t* and axioms are included for each time step up to *t*. If the satisfiability algorithm finds a model, then a plan is extracted by looking at those proposition symbols that refer to actions and are assigned *true* in the model. If no model exists, then the process is repeated with the goal moved one step later.

7.7.4 Making plans by propositional inference

The agent in Figure 7.20 uses logical inference to determine which squares are safe, but uses A* search to make plans. In this section, we show how to make plans by logical inference. The basic idea is very simple:

1. Construct a sentence that includes
    (a) $Init^0$, a collection of assertions about the initial state;
    (b) $Transition^1, \ldots, Transition^t$, the successor-state axioms for all possible actions at each time up to some maximum time *t*;
    (c) the assertion that the goal is achieved at time *t*: $HaveGold^t \wedge ClimbedOut^t$.

2. Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the problem is unsolvable.

3. Assuming a model is found, extract from the model those variables that represent actions and are assigned *true*. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure 7.22. It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps *t*, up to some maximum conceivable plan length T_{max}. In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

The key step in using SATPLAN is the construction of the knowledge base. It might seem, on casual inspection, that the wumpus world axioms in Section 7.7.1 suffice for steps 1(a) and 1(b) above. There is, however, a significant difference between the requirements for entailment (as tested by ASK) and those for satisfiability.

Consider, for example, the agent's location, initially $[1, 1]$, and suppose the agent's unambitious goal is to be in $[2, 1]$ at time 1. The initial knowledge base contains $L_{1,1}^0$ and the goal is $L_{2,1}^1$. Using ASK, we can prove $L_{2,1}^1$ if $Forward^0$ is asserted, and, reassuringly, we cannot

prove $L_{2,1}^1$ if, say, $Shoot^0$ is asserted instead. Now, SATPLAN will find the plan $[Forward^0]$; so far, so good.

Unfortunately, SATPLAN also finds the plan $[Shoot^0]$. How could this be? To find out, we inspect the model that SATPLAN constructs: it includes the assignment $L_{2,1}^0$, that is, the agent can be in $[2, 1]$ at time 1 by being there at time 0 and shooting. One might ask, "Didn't we say the agent is in $[1, 1]$ at time 0?" Yes, we did, but we didn't tell the agent that it can't be in two places at once! For entailment, $L_{2,1}^0$ is unknown and cannot, therefore, be used in a proof; for satisfiability, on the other hand, $L_{2,1}^0$ is unknown and can, therefore, be set to whatever value helps to make the goal true.

SATPLAN is a good debugging tool for knowledge bases because it reveals places where knowledge is missing. In this particular case, we can fix the knowledge base by asserting that, at each time step, the agent is in exactly one location, using a collection of sentences similar to those used to assert the existence of exactly one wumpus. Alternatively, we can assert $\neg L_{x,y}^0$ for all locations other than $[1, 1]$; the successor-state axiom for location takes care of subsequent time steps. The same fixes also work to make sure the agent has one and only one orientation at a time.

SATPLAN has more surprises in store, however. The first is that it finds models with impossible actions, such as shooting with no arrow. To understand why, we need to look more carefully at what the successor-state axioms (such as Equation (7.3)) say about actions whose preconditions are not satisfied. The axioms *do* predict correctly that nothing will happen when such an action is executed (see Exercise 7.SATP), but they do *not* say that the action cannot be executed! To avoid generating plans with illegal actions, we must add **precondition axioms** stating that an action occurrence requires the preconditions to be satisfied.[14] For example, we need to say, for each time t, that

> Precondition axioms

$$Shoot^t \;\Rightarrow\; HaveArrow^t \,.$$

This ensures that if a plan selects the *Shoot* action at any time, it must be the case that the agent has an arrow at that time.

SATPLAN's second surprise is the creation of plans with multiple simultaneous actions. For example, it may come up with a model in which both $Forward^0$ and $Shoot^0$ are true, which is not allowed. To eliminate this problem, we introduce **action exclusion axioms**: for every pair of actions A_i^t and A_j^t we add the axiom

> Action exclusion axiom

$$\neg A_i^t \vee \neg A_j^t \,.$$

It might be pointed out that walking forward and shooting at the same time is not so hard to do, whereas, say, shooting and grabbing at the same time is rather impractical. By imposing action exclusion axioms only on pairs of actions that really do interfere with each other, we can allow for plans that include multiple simultaneous actions—and because SATPLAN finds the shortest legal plan, we can be sure that it will take advantage of this capability.

To summarize, SATPLAN finds models for a sentence containing the initial state, the goal, the successor-state axioms, the precondition axioms, and the action exclusion axioms. It can be shown that this collection of axioms is sufficient, in the sense that there are no longer any spurious "solutions." Any model satisfying the propositional sentence will be a

---

[14] Notice that the addition of precondition axioms means that we need not include preconditions for actions in the successor-state axioms.

valid plan for the original problem. Modern SAT-solving technology makes the approach quite practical. For example, a DPLL-style solver has no difficulty in generating the solution for the wumpus world instance shown in Figure 7.2.

This section has described a declarative approach to agent construction: the agent works by a combination of asserting sentences in the knowledge base and performing logical inference. This approach has some weaknesses hidden in phrases such as "for each time t" and "for each square $[x, y]$." For any practical agent, these phrases have to be implemented by code that generates instances of the general sentence schema automatically for insertion into the knowledge base. For a wumpus world of reasonable size—one comparable to a smallish computer game—we might need a 100×100 board and 1000 time steps, leading to knowledge bases with tens or hundreds of millions of sentences.

Not only does this become rather impractical, but it also illustrates a deeper problem: we know something about the wumpus world—namely, that the "physics" works the same way across all squares and all time steps—that we cannot express directly in the language of propositional logic. To solve this problem, we need a more expressive language, one in which phrases like "for each time t" and "for each square $[x, y]$" can be written in a natural way. First-order logic, described in Chapter 8, is such a language; in first-order logic a wumpus world of any size and duration can be described in about ten logic sentences rather than ten million or ten trillion.

Summary

We have introduced knowledge-based agents and have shown how to define a logic with which such agents can reason about the world. The main points are as follows:

- Intelligent agents need knowledge about the world in order to reach good decisions.
- Knowledge is contained in agents in the form of **sentences** in a **knowledge representation language** that are stored in a **knowledge base**.
- A knowledge-based agent is composed of a knowledge base and an inference mechanism. It operates by storing sentences about the world in its knowledge base, using the inference mechanism to infer new sentences, and using these sentences to decide what action to take.
- A representation language is defined by its **syntax**, which specifies the structure of sentences, and its **semantics**, which defines the **truth** of each sentence in each **possible world** or **model**.
- The relationship of **entailment** between sentences is crucial to our understanding of reasoning. A sentence α entails another sentence β if β is true in all worlds where α is true. Equivalent definitions include the **validity** of the sentence $\alpha \Rightarrow \beta$ and the **unsatisfiability** of the sentence $\alpha \wedge \neg \beta$.
- Inference is the process of deriving new sentences from old ones. **Sound** inference algorithms derive *only* sentences that are entailed; **complete** algorithms derive *all* sentences that are entailed.
- **Propositional logic** is a simple language consisting of **proposition symbols** and **logical connectives**. It can handle propositions that are known to be true, known to be false, or completely unknown.

- The set of possible models, given a fixed propositional vocabulary, is finite, so entailment can be checked by enumerating models. Efficient **model-checking** inference algorithms for propositional logic include backtracking and local search methods and can often solve large problems quickly.
- **Inference rules** are patterns of sound inference that can be used to find proofs. The **resolution** rule yields a complete inference algorithm for knowledge bases that are expressed in **conjunctive normal form**. **Forward chaining** and **backward chaining** are very natural reasoning algorithms for knowledge bases in **Horn form**.
- **Local search** methods such as WALKSAT can be used to find solutions. Such algorithms are sound but not complete.
- Logical **state estimation** involves maintaining a logical sentence that describes the set of possible states consistent with the observation history. Each update step requires inference using the transition model of the environment, which is built from **successor-state axioms** that specify how each **fluent** changes.
- Decisions within a logical agent can be made by SAT solving: finding possible models specifying future action sequences that reach the goal. This approach works only for fully observable or sensorless environments.
- Propositional logic does not scale to environments of unbounded size because it lacks the expressive power to deal concisely with time, space, and universal patterns of relationships among objects.

Bibliographical and Historical Notes

John McCarthy's paper "Programs with Common Sense" (McCarthy, 1958, 1968) promulgated the notion of agents that use logical reasoning to mediate between percepts and actions. It also raised the flag of declarativism, pointing out that telling an agent what it needs to know is an elegant way to build software. Allen Newell's (1982) article "The Knowledge Level" makes the case that rational agents can be described and analyzed at an abstract level defined by the knowledge they possess rather than the programs they run.

Logic itself had its origins in ancient Greek philosophy and mathematics. Plato discussed the syntactic structure of sentences, their truth and falsity, their meaning, and the validity of logical arguments. The first known systematic study of logic was Aristotle's Organon. His **syllogisms** were what we now call inference rules, although they lacked the compositionality $\quad$ Syllogism of our current rules.

The Megarian and Stoic schools began the systematic study of the basic logical connectives in the fifth century BCE. Truth tables are due to Philo of Megara. The Stoics took five basic inference rules as valid without proof, including the rule we now call Modus Ponens. They derived a number of other rules from these five, using, among other principles, the deduction theorem (page 222) and were clearer about proof than was Aristotle (Mates, 1953).

The idea of reducing logical inference to a purely mechanical process is due to Wilhelm Leibniz (1646–1716). George Boole (1847) introduced the first comprehensive and workable system of formal logic in his book *The Mathematical Analysis of Logic*. Boole's logic was closely modeled on the ordinary algebra of real numbers and used substitution of logically equivalent expressions as its primary inference method. Although it didn't handle all of

propositional logic, other mathematicians soon filled in the missing pieces. Schröder (1877) described conjunctive normal form, while Horn form was introduced much later by Alfred Horn (1951). The first comprehensive exposition of modern propositional logic (and first-order logic) is found in Gottlob Frege's (1879) *Begriffschrift* ("Concept Writing" or "Conceptual Notation").

The first mechanical device to carry out logical inferences was the Stanhope Demonstrator, constructed by the third Earl of Stanhope (1753–1816). William Stanley Jevons, one of the mathematicians who extended Boole's work, constructed his "logical piano" in 1869 to do inferences in Boolean logic. An entertaining history of these early mechanical inference devices is given by Martin Gardner (1968). The first computer programs for logical inference were Martin Davis's 1954 program for proofs in Presburger arithmetic (Davis, 1957), and the Logic Theorist of Newell, Shaw, and Simon (1957).

Emil Post (1921) and Ludwig Wittgenstein (1922) independently used truth tables as a method of testing validity of propositional logic sentences. The Davis–Putnam algorithm (Davis and Putnam, 1960) was the first algorithm for propositional resolution, and the improved DPLL backtracking algorithm (Davis *et al.*, 1962) proved to be more efficient. The resolution rule and a proof of its completeness were developed in full generality for first-order logic by J. A. Robinson (1965).

Stephen Cook (1971) showed that deciding satisfiability of a sentence in propositional logic (the SAT problem) is NP-complete. Many subsets of propositional logic are known for which the satisfiability problem is polynomially solvable; Horn clauses are one such subset.

Early investigations showed that DPLL has polynomial average-case complexity for certain natural distributions of problems. Even better, Franco and Paull (1983) showed that the same problems could be solved in *constant* time simply by guessing random assignments. Motivated by the empirical success of local search, Koutsoupias and Papadimitriou (1992) showed that a simple hill-climbing algorithm can solve *almost all* satisfiability problem instances very quickly, suggesting that hard problems are rare. Schöning (1999) exhibited a randomized hill-climbing algorithm whose *worst-case* expected run time on 3-SAT problems is $O(1.333^n)$—still exponential, but substantially faster than previous worst-case bounds. The current record is $O(1.32216^n)$ (Rolf, 2006).

Efficiency gains in propositional solvers have been rapid. Given ten minutes of computing time, the original DPLL algorithm on 1962 hardware could solve only problems with 10 or 15 variables (on a 2019 laptop it would be about 30 variables). By 1995 the SATZ solver (Li and Anbulagan, 1997) could handle 1,000 variables, thanks to optimized data structures for indexing variables. Two crucial contributions were the **watched literal** indexing technique of Zhang and Stickel (1996), which makes unit propagation very efficient, and the introduction of clause (i.e., constraint) learning techniques from the CSP community by Bayardo and Schrag (1997). Using these ideas, and spurred by the prospect of solving industrial-scale circuit verification problems, Moskewicz *et al.* (2001) developed the CHAFF solver, which could handle problems with millions of variables. Beginning in 2002, annual SAT competitions have been held; most of the winning entries have been variants of CHAFF. The landscape of solvers is surveyed by Gomes *et al.* (2008).

Local search algorithms for satisfiability were tried by various authors throughout the 1980s, based on the idea of minimizing the number of unsatisfied clauses (Hansen and Jaumard, 1990). A particularly effective algorithm was developed by Gu (1989) and indepen-

dently by Selman *et al.* (1992), who called it GSAT and showed that it was capable of solving a wide range of very hard problems very quickly. The WALKSAT algorithm described in this chapter is due to Selman *et al.* (1996).

The "phase transition" in satisfiability of random *k*-SAT problems was first observed by Simon and Dubois (1989) and has given rise to a great deal of theoretical and empirical research—due, in part, to the connection to phase transition phenomena in statistical physics. Crawford and Auton (1993) located the 3-SAT transition at a clause/variable ratio of around 4.26, noting that this coincides with a sharp peak in the run time of their SAT solver. Cook and Mitchell (1997) provide an excellent summary of the early literature on the problem. Algorithms such as **survey propagation** (Parisi and Zecchina, 2002; Maneva *et al.*, 2007) take advantage of special properties of random SAT instances near the satisfiability threshold and greatly outperform general SAT solvers on such instances. The current state of theoretical understanding is summarized by Achlioptas (2009).

Good sources for information on satisfiability, both theoretical and practical, include the *Handbook of Satisfiability* (Biere *et al.*, 2009), Donald Knuth's (2015) fascicle on satisfiability, and the regular *International Conferences on Theory and Applications of Satisfiability Testing*, known as SAT.

The idea of building agents with propositional logic can be traced back to the seminal paper of McCulloch and Pitts (1943), which is well known for initiating the field of neural networks, but actually was concerned with the implementation of a Boolean circuit-based agent design in the brain. Stan Rosenschein (Rosenschein, 1985; Kaelbling and Rosenschein, 1990) developed ways to compile circuit-based agents from declarative descriptions of the task environment. Rod Brooks (1986, 1989) demonstrates the effectiveness of circuit-based designs for controlling robots (see Chapter 26). Brooks (1991) argues that circuit-based designs are *all* that is needed for AI—that representation and reasoning are cumbersome, expensive, and unnecessary. In our view, both reasoning and circuits are necessary. Williams *et al.* (2003) describe a hybrid agent—not too different from our wumpus agent—that controls NASA spacecraft, planning sequences of actions and diagnosing and recovering from faults.

The general problem of keeping track of a partially observable environment was introduced for state-based representations in Chapter 4. Its instantiation for propositional representations was studied by Amir and Russell (2003), who identified several classes of environments that admit efficient state-estimation algorithms and showed that for several other classes the problem is intractable. The **temporal-projection** problem, which involves determining what propositions hold true after an action sequence is executed, can be seen as a special case of state estimation with empty percepts. Many authors have studied this problem because of its importance in planning; some important hardness results were established by Liberatore (1997). The idea of representing a belief state with propositions can be traced to Wittgenstein (1922).

The approach to logical state estimation using temporal indexes on propositional variables was proposed by Kautz and Selman (1992). Later generations of SATPLAN were able to take advantage of the advances in SAT solvers and remain among the most effective ways of solving difficult planning problems (Kautz, 2006).

The **frame problem** was first recognized by McCarthy and Hayes (1969). Many researchers considered the problem unsolvable within first-order logic, and it spurred a great deal of research into nonmonotonic logics. Philosophers from Dreyfus (1972) to Crockett

Survey propagation

Temporal-projection

(1994) have cited the frame problem as one symptom of the inevitable failure of the entire AI enterprise. The solution of the frame problem with successor-state axioms is due to Ray Reiter (1991). Thielscher (1999) identifies the inferential frame problem as a separate idea and provides a solution. In retrospect, one can see that Rosenschein's (1985) agents were using circuits that implemented successor-state axioms, but Rosenschein did not notice that the frame problem was thereby largely solved.

Modern propositional solvers have been applied to a variety of industrial applications, such as the synthesis of computer hardware (Nowick *et al.*, 1993). The SATMC satisfiability checker was used to detect a previously unknown vulnerability in a Web browser sign-on protocol (Armando *et al.*, 2008).

The wumpus world was invented as a game by Gregory Yob (1975). Ironically, Yob developed it because he was bored with games played on a rectangular grid: he put his wumpus on a dodecahedron, and we put it back onto the boring old grid. Michael Genesereth suggested that the wumpus world be used as an agent testbed.